**Towards a Multi-domain Software-defined Network Controller**

Rajesh Gopidi

UNC Chapel Hill

**ABSTRACT**

In this paper we address the problem of resource delegation in software-defined networks, especially in software-defined data center networks and Wide Area Networks (WANs). One of the key limitations of the existing delegation/virtualization frameworks is, only a partial information about the delegated resources such as basic connectivity between the nodes in the network is shared with the tenants, leaving behind some of the crucial details – like bandwidth allocated on an interface. In addition, control of the tenants over delegated resources is limited, thus inhibiting them from subleasing unused resources, or in some cases a subset of acquired resources -- like integrating delegations from multiple providers and delegating a part of it-- to other tenants for profits. To address these problems, we present the overall design of our resource delegation framework in which the information about the delegated resources is explicitly communicated to the tenant's controller in a finer detail -- including flowspace available on various links and switching constraints applicable to the switches in the virtual network -- and is given complete control over them, allowing it to recursively delegate them. Furthermore, we describe the architecture of two types of controllers used in our framework. We also built a prototype of our framework to prove efficacy of our design and bench marked our topology validation process against the traditional Link Layer based topology discovery method, LLDP.

**1. Introduction**

Cloud computing is becoming the norm in the industry as most of the small and medium scale organizations are more than interested in hosting their services in the cloud to benefit from its flexibility to quickly enable various network services and its lucrative pay-as-you-go model which can result in considerable cost savings, as it eliminates server maintenance, power and cooling costs from their budget. Infrastructure-as-a-service (IaaS) cloud model, in particular, is gaining traction, as it caters to a wide range of audience by allowing them to build and configure their own virtual computing infrastructure by renting computing platform resources in the form of virtual machines [1]. Furthermore, this model is of importance in academic world too, as projects like GENI in USA [2] and OFELIA in Europe [3] enable researchers to test new solutions in large–scale realistic environments. Virtualization of network resources, both nodes and links, enables providers of such model to provision multiple tenants and isolate them from each other while they co-exist. Thus, increasing their revenues.

SDN, especially OpenFlow has been widely adapted in the datacenter environment by the providers to offer such models to end users. For instance, FlowVisor [4] and its descendents (ADVisor [5] and VeRTIGO [6]) that leverage on the hardware abstraction provided by OpenFlow [7] are deployed to share a hardware forwarding plane among multiple logical networks. It is noted that, in the existing frameworks, users are either presented with a simple abstraction of the underlying network also known as "the Big Switch approach," where all the virtual machines allocated to a user can reach each other or in some cases they are given programmatic control at the network edge, to enable improved access control. This lack of transparency in network flow configuration and unstable Service Level Agreements (SLAs) makes the cloud model less attractive.

In addition to datacenters, SDN has its presence in WANs connecting multiple data centers and multi-domain networks of a few large providers [8]. Most of the research related to SDN in WANs is concentrated around the placement of controllers [9] and the existence of multiple controllers, one for each domain, to offer services across domains [10]. While the rest of the SDN efforts are centered on virtualization of the service provider network consisting of multiple technology layers and multiple domains as a single big switch [11, 12]. However, we believe none of the existing frameworks exploit the complete potential of the SDN concept to simplify the role of Internet Service Providers by introducing new stakeholders (virtual network providers) into the ecosystem as described in [13].

To address the above described problems, in this paper, we describe a part of our ongoing work, a resource delegation framework for multi-domain software-defined networks in which the infrastructure provider can dynamically delegate network resources to the connectivity provider, who on the other side, can build his own network by stitching delegations from multiple providers and provision multiple service providers on top of them. In our framework, information about the delegated resources -- includes the flowspace on all the links part of the delegated network topology and switching constraints on various switches-- is stored in a DEX graph database format [14] and given to the connectivity provider's OpenFlow controller, which is capable of parsing the graph and validating the delegations. In this way, the connectivity provider can build his own network from delegations in multiple domains and acquires complete control over it. Furthermore, our framework can be deployed in the cloud leading to a specific case where there is only one infrastructure provider and the tenant plays the role of network provider who gains full access to the test bed.

**2. Background**

A. Software Defined Networking (SDN)

SDN is a relatively new but a revolutionary approach to computer networking that simplifies the task of managing a network by abstracting vendor-specific details of the forwarding devices. This is achieved by decoupling the entity that is responsible for computing routes (the control plane) from the underlying hardware that forwards the traffic along precomputed paths (the data plane). As a result, the control plane is logically centralized into an SDN controller running on a cluster of machines or in some cases a single server.. SDN enables network administrators to monitor and configure the network from a central location, unlike in traditional networks, where there is a high possibility of an error due to manual configuration of switches. It also accelerates innovation in the control plane, as it is easy to evaluate new protocols in a logically centralized control system rather than a distributed one. In SDN, the control plane (controller) and the data plane (switches) communicate with each other via a vendor agnostic API, not necessarily OpenFlow.

B. OpenFlow

OpenFlow is one of the most popular API used in software-defined networks to manage switches through a controller running on an external device. In OpenFlow framework, the forwarding architecture of a switch is abstracted in the form of one or more flow tables -- a sequence of flow tables is used to represent some of the complicated operations such as, IP Fast re-route [15] performed by the switch. Using the OpenFlow protocol, the controller can add, modify and delete entries in the flow table. Each flow table contains a set of flow entries; each flow entry consists of a set of match fields, an ordered list of instructions to apply to packets that match the entry and counters to keep track of statistics. OpenFlow controller computes the paths either re-actively or proactively, and translates them into flow entries to be installed in the switches along the paths, with a fixed life span. For the duration of the life span the set of packets matching the flow entry -- also called a flow -- are forwarded by the switch at line rate. All the non-matching packets are either forwarded to the controller or dropped by the switch. This again depends on the configuration of the flow table managed by the controller. Furthermore, to improve reliability, OpenFlow allows switches to connect to multiple controllers. Out of all connected controllers, only one of them plays the master role, which implies, full access to the switch while the rest of the controllers are in standby mode with read-only access. However, in its latest specification, OpenFlow protocol enables more than one controller to program a switch simultaneously.

**3. Related Work**

In the last few years, a fair amount of literature related to virtualization and/or delegation of network resources has been published, with SDN as the cornerstone. Out of all, Flowvisor has led the way, and has been the most noted one. Flowvisor stems from the idea of virtualization of computer hardware. The introduction of OS hypervisor resulted in rapid development of both, hardware and software, as it allowed to limit/constrain direct dependencies between hardware and software. Flowvisor follows a similar path in virtualizing networks, where, traditionally, there has been no complete network hardware abstraction. While MPLS can slice forwarding tables by allocating a set of labels and VLANs can slice the link layer, no technology prior to SDN virtualized a complete network. Moreover, with the present software stack in the deployed network hardware, it is almost impossible to abstract the hardware -- switches/routers from different vendors have proprietary architectures/forwarding pipelines. As a result, Flowvisor builds on top of the OpenFlow protocol as a common hardware abstraction layer, and places itself in-between the control and the data plane of a switch. It uses the OpenFlow protocol to communicate with the OpenFlow agent running on top of the switch, which then translates OpenFlow messages to corresponding hardware level instructions to be executed by the switch processor. This is similar to how a type 2 hypervisor running on top of a host OS intercepts system calls from guest OS and maps them to corresponding host OS system calls, to coordinate access to the underlying hardware.

As stated in section [2], an OpenFlow switch can attribute the master role to only one controller; there can be multiple secondary controllers connected to the switch with read-only access, allowing one of them to take control of the switch with very little session startup time, on failure of the master controller. However, in order to slice the network and have each slice managed by a different controller, multiple controllers should be able to control a switch simultaneously unless the size of the network is so large that each switch is not shared by any two slices. Flowvisor solves this particular problem by acting as a proxy for all the controllers authorized to program a switch. With multiple controllers -- one controller per slice -- hosted on top of it, Flowvisor forwards the message(s) received from a switch to the respective controller(s). In other words, all the messages sent from OpenFlow-enabled network devices to multiple host controllers and vice-versa, pass through Flowvisor.

In Flowvisor, an instance of virtual network or a slice[[1]](#footnote-2) is defined as an ordered list of tuples -- or rules, if we may call it for simplicity -- each consisting of a particular switch and port ID, and the expected

packet header, which is represented by 10 fields of the header (from the source MAC address to TCP port numbers). Except for the Switch ID, the rest of the fields can be substituted with wildcards, thus providing fine grain control over slicing of the network. Packet header fields included in slice definition are exactly the ones supported by OpenFlow flow entries that match on any subset of the bits in the 10 fields of the packet header. A flow match on given 256 bits making up to 10 fields in the packet header, is same as a point in 256-dimensional geometric space where as a match with *k* wildcard bits represents a volume in the same space [16]. Thus, a slice defined as a set of tuples is in reality a set of possible flows occupying (most likely non-contiguous) volume(s) in the geometric space; which the authors call it as slice's flowspace.

Flowvisor also acts as a transparent rule enforcer to ensure isolation between two or more virtual networks. Given a packet at any point in the network, by examining the flowspace definitions of all the slices, flowvisor is capable of determining the slice/flowspace the packet belongs to. Furthermore, whenever flowvisor receives a packet from the guest controller, it runs through the rules in the controller's flowspace, in the specified order and forwards the packet unchanged if the message acts only on traffic within its flowspace. Otherwise, either the message gets translated into a more flowspace specific message (subset of the earlier flowspace) and is sent to the switch, or is even bounced back to the controller in the form of an OpenFlow error message. Similarly, a message from a switch is examined to determine the relevant network slice(s) and is forwarded to the respective guest controller(s). In addition to the header fields, rules in slice definition also include action attributes such as allow, deny, read-only. Read-only action allows a controller to receive Open flow control messages and query for switch statistics, but prevents it from programming the switch, when operating on flowspace represented by the tuple.

While Flowvisor is able to solve the problem of network substrate virtualization, some of its crucial features are left for future work:

1) Support for true virtual topologies: a virtual network topology has to be a subset of the physical topology, leaving no choice to the network provider to simplify the slice by introducing virtual link(s) -- path(s) between two switches comprising of multiple links -- and hiding some of the switches. For instance, consider the physical and desired virtual topologies shown in the below figure. The desired topology is not possible with Flowvisor's architecture because, it lacks the intelligence to hide the switch 'C' from tenant's controller and directly link switch A with E and G by dynamically programming necessary flow rules in switch 'C' so that user traffic on the two virtual links is uninterrupted.

2) Virtual Flowspace: Presently, there is no way two slices can simultaneously share the same flowspace and operate without interfering with each other's traffic. For example, two slices might want control over the same desirable flowspace, say, the 10.0.0.0/8 IP netblock. This problem is similar to the case of two applications running on top of an Operating System using identical memory address space to access different content in memory. And it can be solved in a manner similar to virtual memory by allocating same flowspace to multiple tenants, and transparently rewriting it to non-intersecting regions.

3) Device Configuration: Guest controllers are allowed to program the forwarding table in switches ; however, they are not permitted to change the switch configuration or, for example, enable features such as low-power transmission mode, change the advertized Extended Service Set Identification (SSID), etc. in wireless devices. However, as stated in [4], inclusion of a device configuration protocol in OpenFlow can solve this particular problem.

Authors in [5] present a system called ADVisor (Advanced FlowVisor) -- an extension of Flowvisor to overcome the first two of the above stated challenges. Flowvisor is at the core of the system, and is responsible for network slicing and providing an interface to OpenFlow switches and controllers while three additional modules: 1) Topology Monitor 2) Link Broker 3) Port Mapper enable ADVisor to

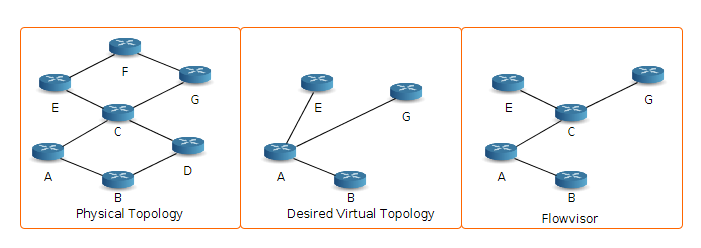


Figure [1]: Flowvisor based networking virtualization

manage virtual links. Unlike in Flowvisor's architecture, flowspace of a slice is defined by a combination of bits from Open Systems Interconnection (OSI) - layer 2 fields of the packet header. In other words, flowspace is a set of all packet headers with a specific u bits, not necessarily contiguous, being common in all the L2 headers. The value of the specific *u* bits is referred to as *slice\_tag*. Apart from the slice tag, ADVisor's slice definition includes a subset of switches in the physical network, a set of access ports on switches in the subset, a set of virtual links and a set of virtual ports connecting virtual links to the switches. Out of u bits of the slice\_tag, v bits are used to identify a virtual topology (*slice\_tag\_si*) i.e., all the packets of the virtual topology are tagged with the same *slice\_tag\_s*i and Flowvisor uses it to slice the network; the rest v bits (*slice\_tag\_vl*) are used to identify virtual links within the virtual topology. Whenever ADVisor receives a Packet-In message from a switch, the topology monitor checks the configuration files to find the slice\_tag\_si associated with the switch port and writes it into the message if not already present; however, it is not clear from the description in the paper, if the guest controller is made aware of the slice\_tag\_si, as there is no mention of removing the tag before the packet is forwarded to the controller, and re-inserting it after receiving a reply from the controller. Topology monitor is also responsible for determining whether to forward the message to Flowvisor or Link Broker based on the port type -- virtual or real. The Link Broker's main objective is to control switches and ports part of virtual links. The slice controller programs the end points of the virtual links; Port Mapper programs the hidden switches based on the slice\_tag\_si and slice\_tag\_vl identifiers in the packet so that flows on virtual links are uninterrupted. Port Mapper intercepts all Openflow messages originating from either side -- message from switches as well as from the slice controllers -- and remaps the virtual ports to the corresponding physical ports and vice-versa. Overall, ADVisor is able to enable real virtual topologies in SDN based networks, but not without some major drawbacks. Usage of only OSI-layer2 fields as the slice identifiers limits the scalability of the solution. Moreover, introduction of a tag (example: MPLS or VLAN tag) on every packet of a virtual topology results in additional overhead and increases latency proportional to the number of tenants.

With ADVisor as a precursor, the same group of authors as in [5], propose VeRTIGO (ViRtual TopologIes Generalization in OpenFlow networks) [6], an SDN-based platform designed for network virtualization. VeRTIGO's architecture, similar to ADVisor, is developed with Flowvisor as the centerpiece; however, it differs significantly from ADVisor, as it exposes different views of the network to different controllers: 1) A fully connected virtual network with complete control over it; 2) A single abstract node with access ports being virtual ports and all the other details of the topology hidden from the controller. Furthermore, the virtual links are no longer distinguished based on the slice\_tag\_vl identifiers, instead they propose the use of a database to store packet headers of each flow traversing virtual links. Instantiation of a fully connected virtual network is same as described in case of ADVisor but, it is not clear from the paper if the flowspace is still defined as a subset of the bits in the OSI-L2 layer fields. To provide a single abstract node view of the virtual topology, VeRTIGO replaces Topology Monitor and Link Broker in ADVisor with Classifier and Internal Controller modules; besides, introduces three new modules: 1) VT Planner, 2) Node Virtualizer and 3) Storage. The Classifier module is responsible for classifying messages received from a switch before forwarding them to the appropriate controller. If a message originated from endpoints of a virtual link or access ports, it is sent to the slice's controller. In all other cases, it is forwarded to the internal controller, which handles it by taking necessary actions in the form of programming flow entries so that flows are uninterrupted. The role of VT planner is to map virtual links to physical resources when a new virtual topology is created, changes are introduced into the physical topology or when unexpected events occur in the network. VT planner implements a path selection algorithm which is fed with real-time traffic load statistics to derive the mapping of virtual links to actual links in the network. The storage module exports an interface to store configuration files of slices, and to handle the database of packet headers of flows traversing virtual links. Storing packet headers instead of adding/modifying a tag in the header scales well, but the authors fail to address the case where two flows with the same header sequence traversing two virtual links composed of a common physical link are isolated from one another.

Although VeRTIGO enables network virtualization in two extreme forms, scalability of the architecture is questionable, especially in the data center environment. For every virtual switch, the virtualization layer has to initiate a TCP session and exchange OpenFlow messages with the guest controller. The communication overhead and latency on the control channel is not significant for supporting a single tenant, but with multiple tenants it quickly proliferates, as the virtual switches outnumber the physical switches at least by an order of magnitude. To overcome the scalability challenge, the authors in [17] propose FlowN, an efficient and scalable virtualization solution inspired from container-based virtualization. Similar to VeRTIGO, FlowN supports multiple abstractions ranging from arbitrary virtual topologies to a simple "one big switch" abstraction. FlowN, however, does not employ Flowvisor to slice the network; instead, it completely decouples the virtual topology from the physical infrastructure. As a result, each tenant gets to design its own network and include the resource constraints such as number of flow table entries in a switch; the maximum latency and bandwidth of a link. The virtual topology is then mapped to the physical infrastructure by the provider, by running an embedding algorithm [18]. Thus, FlowN virtualizes the flowspace similar to how an operating system virtualizes memory. To distinguish traffic and flow rules of tenants, FlowN encapsulates tenant's traffic with a protocol agnostic header (for example, VLAN header) transparent to the tenant's controller and end hosts, at the edge. Whenever FlowN receives a packet from a switch, it classifies the packet based on: 1) label in the encapsulation header 2) physical port 3) fields specified by the tenant application. After the classification phase, the physical components are mapped to corresponding virtual components based on the config stored in a relational database. Similarly, virtual-physical mapping occurs when a tenant's controller sends an OpenFlow message. Switches swap labels in the encapsulation header throughout the network, to support a large number of tenants. Nonetheless, with large number of tenants, the mapping between the virtual and physical flowpsace can easily become the bottleneck. To overcome this bottleneck, FlowN virtualization layer is decentralized by partitioning it across multiple servers, with each server hosting a replica of the database -- one of them is elected as the master and the rest are in stand-by mode.

Unlike the previous virtualization frameworks, FlowN in itself is a controller that runs tenants' controllers in the form of applications, each with its own address space, virtual topology, and event handlers. Rather than mapping OpenFlow messages, FlowN maps between the controller's API calls. In essence, FlowN is a transparent layer between the controller and the application(s) that maps events to appropriate event handlers of the tenant's application(s). For example: when a packet-in message is received, FlowN's event handler gets notified. The handler identifies the appropriate application(s) and calls the respective tenant-specific event handler after translating the packet header into the tenant's flowspace. Similarly, FlowN intercepts every API call invoked by the tenant's event handler, and maps between the virtual and physical address space. To avoid a single point of failure, each tenant's controller is run in a separate thread so that failure of one tenant's controller doesn't affect other tenants. But, this type of framework restricts the tenants from running a multi-threaded independent controller on a multi-core server without any interdependencies with the provider's controller -- which is the case in container-based virtualizations. Moreover, this may also hinder tenants from rapidly introducing new features into their networks, as they wait for the provider to update its API.

Elaborating on their previous work, the authors of [19] present RouteFlow: a commodity routing framework based on SDN paradigm that combines logically centralized control with line-rate performance of the hardware. RouteFlow centralizes IP routing services, albeit in a logically distributed fashion (across several servers), thus improving the flexibility of IP networks and enabling rapid innovation and customization of routing protocols and algorithms. RouteFlow architecture is composed of: an OpenFlow Controller, an interface to send and receive messages from OpenFlow-enabled switches; a RouteFlow server (RF-S) responsible for virtualizing and interconnecting the IP routing engines, which otherwise run on switches in legacy networks. For every switch discovered by the OpenFlow network controller (NC), RouteFlow controller (RF-C), an application on top of NC, relays the message to the RF-S. The server then spins up a virtual machine (VM), which runs a stack of open source routing protocols (e.g., OSPF, BGP). Each VM is configured with the same number of virtual NICs as the number of active ports on the corresponding OpenFlow switch. Moreover, VMs are dynamically interconnected via software switches to form a logical topology that is a subset of the physical network.

Once the virtual network is set up, the routing protocols running in the vms exchange control packets -- this can be achieved in two ways: packets can be sent all the way down to the physical switch and back up, or the packets can be forwarded through the software switches. Each individual VM, according to the configured routing protocols, computes the Forwarding Information Base (FIB), which is then sent as an update message to the RF-Server. The RF-Server then translates the FIB into corresponding OpenFlow rules and requests the RF-Controller to install them into the switch's forwarding table. To transparently integrate legacy networks, RF-Server uses flow entires to match the routing protocol-specific control packets and forwards them to the corresponding virtual instances. Conversely, it also relays the pertinent routing messages to the physical entities. Apart from compatibility with legacy networks, RouteFlow improves the flexibility and operational costs of the provider by supporting three modes of operation: 1) Logical Split: Every physical switch is mapped to a virtual routing engine; 2) Multiplexing: In this mode, multiple virtual routing engines are mapped to the same physical switch (n to 1), thus supporting multi-tenant virtual networks; 3) Aggregation: Multiple physical switches are bundled into a group and mapped to a single virtual instance. This simplifies the network management and signaling scalability, as the neighboring devices can treat the aggregated as a single element. These three modes virtualize network resources, but not to the full extent as they restrict themselves to L3 routing services. Moreover, installing the entire FIB into the switch's forwarding table doesn't help in reducing the costs of the provider, as larger tables are required to support multiple tenants. With multiple processing elements in the slow path (NC, RF-C, RF-S, VM), packets without flow entries experience higher delays, thus affecting the flow setup time and its throughput, especially in case of smaller flows. Lastly, as stated in the future work section, with all the components lacking resilience in case of failure, RouteFlow's architecture suffers from a single point of failure.

Modern wide area networks or networks in general, consist of multiple domains, and span across multiple data centers, enterprise networks and customer sites. Authors in [10] propose a framework titled, DISCO: Distributed Multi-domain SDN Controllers, to architect such multi-domain heterogeneous networks interconnected with a large variety of network technologies ranging from high-capacity leased lines to limited-bandwidth satellite links, or from costly but highly secured links to cheap but unsecured ones. In DISCO's framework, a single controller is responsible for all the intra-domain traffic, and exchanges aggregated domain-wide information with other domain controllers using a lightweight east-west API for end to end flow management. DISCO's architecture is primarily composed of two parts: an intra-domain part, responsible for a single domain's network state, and an inter-domain part, which manages the communication with other controllers. In addition to the core modules (Switch manager, Host manager and Link discovery) imported from the Floodlight controller [20], the intra-domain part is made up of four important modules. The Extended Database module is the central component in which the controller stores the network state, both intra and inter domain, to be used by other modules to compute paths and enable end to end flow management. Information about the individual flows within the domain is gathered by the Monitor manager, using OpenFlow protocol. In addition, it also determines the effects of congestion in the network, by tunneling packets through different paths within the domain at regular intervals. The latency and packet loss on inter domain links is also measured by a initiating a simple ping. The path computation module uses this information to compute routes for new flows and preempts existing low priority flows, if necessary. Finally, the Service manager module is responsible for enforcement of SLAs inside the network.

The inter-domain functionality consists of two key elements: messenger module and Agents, which rely on the messenger to communicate with other controllers. The messenger module implements a lightweight control channel between neighboring domains. It aids the controller in discovering other domain controllers and in exchanging status information (link state, host presence) as well as requesting actions (e.g., reservations) from them. The messenger is implemented as a Floodlight application module with RabbitMQ [21], a broadcast enabled version of the Advanced Message Queuing Protocol (AMQP) [22], as a base to support both point to point and group communications between controllers. It offers a publish/subscribe communication channel with two special topics for inter domain exchanges between agents. First of the two topics named, ID.\*.\* (ID the identifier of the controller) is used by other controllers to send messages directly to it. The other topic, general.\*.\* allows the controllers to broadcast messages to all the other controllers. Apart from these two topics, agents are allowed to define new topics to be subscribed to by other controllers, to receive messages.

To support flows across domains, four main agents -- Monitoring, Reachability, Connectivity and Reservation – are implemented in DISCO's architecture. Connectivity agent is responsible for advertising to other domains the presence of inter-domain links and changes in them over time, in an event driven fashion. The information published by the connectivity agent, like any other agent, is updated in the extended database, and is used by the path computation module to compute routes spanning multiple domains. While the monitoring agent periodically sends aggregated network state such as available bandwidth between all pairs of edge switches/routers, reachability agent notifies on an event basis, the presence of hosts in the domain to other domains so that new/existing paths can be computed/modified to reach the hosts. Lastly, the reservation agent, similar to Resource Reservation Protocol (RSVP), reserves bandwidth for inter-domain flows by directly communicating with respective domains. The service manager in each domain handles the request by allocating required bandwidth. The reservation agent is also responsible to notify other domains to free the bandwidth allocated to an inactive flow.

Though the architecture of DISCO appears to be pertinent, at least theoretically, to multi-domain heterogeneous networks, the scale of the test bed used to evaluate the framework does not help in asserting it. With the in-band control channel, network footprint of the control information increases with the size of the network. However, to avoid it, the authors propose a reduction in frequency of control messages, which in turn, might result in incorrect path computations because of the inaccurate network state. Moreover, at a high incoming rate of inter-domain flows, the set-up time of a flow increases with scale, because of the coordination required between domain controllers to configure a flow. Lastly, with extended database being a shared resource, multiple concurrent reads and writes increases the flow set-up time because of the additional delays experienced by reads/writes.

As Cloud Computing and SDN become pervasive, in the paper [23], the authors bring to surface an argument that cloud users, especially of Infrastructure- as-a-service (IaaS) cloud model, should no longer be provided with a virtualized view and a limited control over of the network. Instead, they propose delegation of some of the network controls to end users by introducing an SDN-based framework. However, the authors acknowledge the downside of such a framework -- exposing too many details of the underlying network can raise security concerns for the provider-- and suggest ways to balance them against the level of control given to the users. In the proposed framework, the user is given a logical view of the network, which is composed by removing a subset of physical switches from the computed topology, and substituting the rest of them with logical switches. In this way, the user is given partial control over flows through each physical switch in the network. Moreover, to enable programming of flow rules by the user's OpenFlow controller, the framework consists of a translation layer between the controller and the network that maps the messages from the controller to operations that modify the physical switches.

The authors associate several properties with virtual views and categorize them into three groups: local properties, strictly dependent on the one to one mapping between the logical and physical network; global properties, defined over two or more delegated views; and security properties. Few examples of local properties stated in the article are: Connectivity property, if the view allows the user's controller to program flows between all of its virtual machines; Geographic consistency is satisfied by a view, if the networks in different geographic locations are mapped to different virtual switches, enabling the controller to take into account long delays experienced on the links connecting those virtual switches; Datacenter consistency is satisfied, if geographic consistency is fulfilled at the datacenter level. Non-interference and fault isolation are two examples of global properties. Two views are said to be non-interfering if actions defined in one view do not affect the other view. Fault isolation property ensures that a fault in one view is segregated from other virtual networks. Security properties are defined as a combination of global and local properties. For example: Non-interference property is defined as a combination of a global property which ensures the union of cloud users' views to be a proper subset of the entire network and a local property which ensures that the maximal subset of switches is exposed to each user out of all possible subsets. Lastly, the authors present two possible instantiations of the framework using Flowvisor and ADVisor, and describe the implementation of Flowvisor based prototype, which uses standard Flowvisor with custom user controllers hosted on top of it. Nonetheless, it is not clear from the prototype description, the contribution of the authors apart from adding a custom flow programming module to POX controller [24], whose functionality is available in most of the open source OpenFlow controllers.

In all of the proposed virtualization/delegation frameworks, very little information about the delegated resources is communicated to the user's controller. However, to operate seamlessly, it is important that the user's controller is aware of the flow space allocated to it, especially in multi-domain networks, where there is a possibility of mismatch between allocated flowspaces in two different domains, because of flowspace fragmentation.. And it is also crucial that the framework supports dynamic resource delegations. In essence, the provider should be able to notify the user's controller to renounce some of the unused resources or update its network by including a new set of delegated resources. Furthermore, for efficient usage of delegated network resources such as bandwidth, hardware capabilities of the switch, the user's controller must be explicitly told about them, which is not the case in the existing frameworks. To address these problems, in this paper, we present a part of our ongoing work, A Resource Delegation Framework for Multi-domain Software Defined Networks in which the information about the delegated resources is explicitly communicated to the users' as well as the domain controllers. Our framework also enables user controllers to stitch delegations from multiple domains controlled by different providers.

**4. Resource Delegation Framework**

In this section, we present our framework for resource delegation in software defined networks, especially for multi-domain networks. We describe the framework in three phases: First, we start with an overall design of the framework[[2]](#footnote-3) that enables a user to seamlessly build a new network by leasing resources from different providers and achieve complete control over it. We then list the essential components that make up the user controller -- or the transit controller – along with the description of their functionality. Lastly, we discuss how the domain controller delegates resources to users and modifies its flowspace accordingly.

**4.1 Overall Design**

The key components of our framework are shown in figure [2]. It consists of a Flow Space Manager (FSM) per domain, which is responsible for isolation between network slices in terms of: Bandwidth, to ensure that a slice's traffic does not exceed the allocated bandwidth at any point in the network; Switch CPU, FSM should track the CPU usage of different slices, and ensure a fair share is allocated to each of them; Flowspace, each slice's controller must be restricted to only programming flows within its own flowspace. As seen in the figure, FSM interacts with the controllers and the underlying hardware via OpenFlow protocol. There are a number of OpenFlow controllers that are similar in functionality to FSM, but we employ Flowvisor in our framework, as it has been widely deployed and is compatible with a wide variety of controllers out there. However, there are some shortcomings of Flowvisor, for example, absence of a secure channel to create new slices by the domain controller or Resource Manager, which we do not discuss here as it is out of scope of this paper. All the controllers -- domain as well as the transit -- connect to the FSM, which relays messages to the switches connected to it and vice-versa.

Each domain hosts a Resource Manager (RM), which monitors the usage of resources by the set of controllers consisting of both domain and transit controllers. In order to monitor the network, the RM establishes a TCP connection with the FSM and periodically retrieves different statistics like the flow statistics, port statistics etc. With all this information and the knowledge of delegated resources at its disposal, RM fulfills new requests for resources by expressing it as topology embedding problem, which is not described here as it is out of scope of this paper. In case the RM is not able to satisfy the request, it reclaims some of the unused resources from other users, if there are any, and fulfills the request. The newly computed virtual topology is communicated to the domain and the transit controller through independent channels in the form of a serialized DEX graph database. We chose DEX graph database to store and retrieve topology because of its ability to handle large sized graphs and its support for labeling of nodes and links. A node in the graph represents a unique port in the virtual network, and an edge represents a unidirectional link -- either virtual or an actual link. Different types of ports and links are differentiated based on the attributes and labels associated with them. The domain controller, after reconstructing the virtual network from the serialized file, flags the delegated resources as unusable by the hosts local to the domain, and notifies the FSM to create a new slice with the flowspace and the URL of the controller set to the ones specified in the request.

Next, we discuss the architecture of the transit controller and the functionality of different modules embedded in it.

**4.2 Transit Controller**

The key building blocks of the transit controller are shown in figure [3]. As seen there, the transit controller is implemented around the core modules of the Floodlight controller. The core modules: Topology manager, Device manager and the FloodlightProvider module, implement some of the fundamental network services a software defined network would expose to its applications. The FloodlightProvider module handles connections to switches and turns OpenFlow messages into events, only to be delivered to other interested modules. The Topology manager computes the topology for the controller, based on the link information received from the Topology validation module, which is discussed later in the paper. It subscribes to various events such as PacketIn, FlowRemoved and PortStatus, notified by the FloodlightProvider and alters the topology before dispatching update messages to other modules. All the end hosts in the network are tracked and classified by the Device manager as they move around a network. Device manager, by default, distinguishes devices based on the MAC and the VLAN tag.

Our architecture's capability to support dynamic delegation of resources is centered around two modules, namely, GraphDB Reader and Topology validation. These two modules help the controller in

reconstructing the topology from the serialized DEX graph file(s), and verifying the validity of the

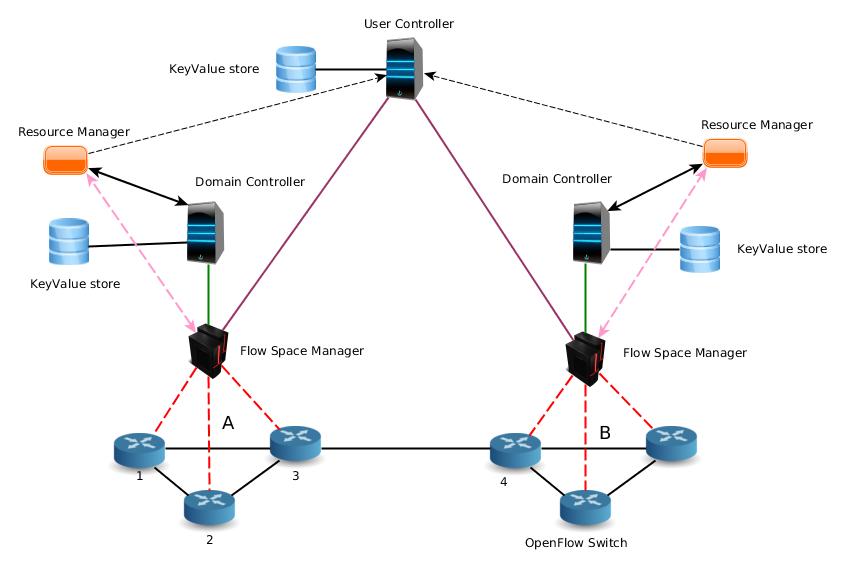


Figure [2]: Overall view of different components in the framework

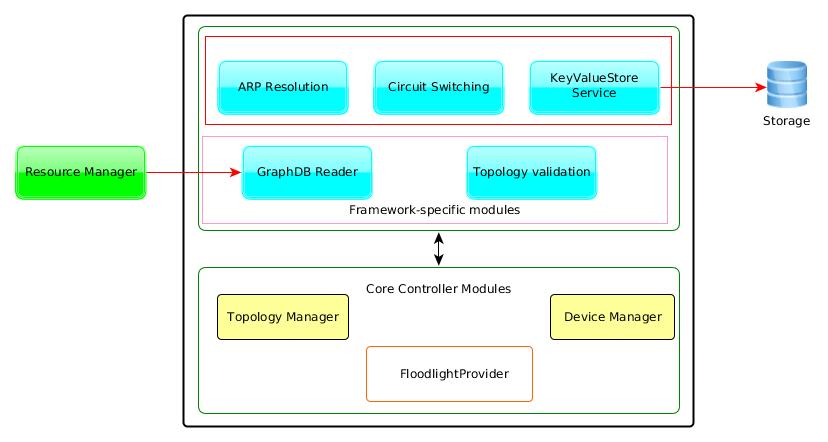
flowspace delegated to it. The rest of the modules which are common between the transit and the domain controllers, allow them to operate in an efficient, scalable and fault-tolerant manner.

**GraphDB Reader Module:** This module is implemented as any other Floodlight controller module, extending a generic interface so that the provider/user can plug in their own versions into the controller by implementing the generic interface. The module listens on a specific port (6999) to receive new delegations from the RM in the form of an RPC request with DEX-based representation of a topology as one of the arguments. On receiving the serialized DEX graph database, the module starts parsing the file and retrieves the topology on a link by link basis. As of now, we support two types of links in the framework. This module distinguishes between the two types based on the labels, "Can be connected to" or "connected to". The "can be connected to" label signifies a virtual link and it is primarily used to represent the possible connection between two ports inside the switching fabric of a switch. It is quite possible that two ports of a switch cannot be interlinked because of flowspace mismatch between them and lack of switch's label translation capability, which is also communicated through attributes associated with an edge. In order to retain this information, we use a matrix-like data structure that maps connectivity between different ports. The label translation capabilities of the switch are stored in a separate integer field. The other label, "connected to" indicates a point-to-point link in the virtual topology, while in reality, the link can be a virtual one representing an aggregation of several physical links in the physical network. The "Connected to" links have an attribute called "Rules". This attribute is non-empty for a virtual link with flowspace mismatch between its two end points. Entries in "Rules" map a point in flowspace of the link's tail end to a unique point in the other flowspace. Within the controller, Rules are stored in a data structure which emulates the forwarding table inside a switch. Apart from the two types of links, the graph is also composed of inter-domain links which are distinguished based on the values of the nodes' domain attribute.

As the module gradually parses the serialized graph, it creates, what we call a shadow topology, where each port is attributed with flowspace, both in ingress and egress directions. Flowspace data structure in the controller consists of discrete data structures for each field in the packet header, starting from the source MAC address to the TCP destination port. It is implemented in such a way that it enables the controller to verify the existence of a delegated resource, and also to pick a random point in the flowspace volume allocated to it. For example, a synchronized sorted tree map is used to represent the VLAN tags. This implementation also supports a way to specify a range of values such as, VLAN, 0 -100. It also allows the controller to specify resources delegated to other users by associating a flag with every delegated flowspace volume. This way of representing the flowspace rather than using a big integer field to encode the packet header, makes the operations on the flowspace volumes faster, as a full word is processed at a time, instead of a bit. Once the shadow topology is completely recreated from the graph, we place the topology into a queue to validate it later, and wait for the switches to announce themselves to the controller, as there is no way a controller can connect to a switch using OpenFlow protocol.

**Topology Validation module:** this module verifies the accuracy of the delegated topology by dispatching probes throughout the network. Although it is completely acceptable to put the network into production without verifying the connectivity between the nodes, we believe it is important to confirm whether the configuration of the underlying topology is in line with the resources delegated to the user. For example, consider the topology in the figure [2], say if the provider hides the switch 2 from the user and replaces it with a virtual link between switch 1 and 3 in the topology, successful transmission of packets with distinct headers from one end to the other end of the virtual link is conditioned by the accuracy of the configuration of the forwarding table in switch 2. One way to test the connectivity is to apply vendor-neutral link layer protocols like LLDP (Link Layer Discovery Protocol), which is typically used by most of the OpenFlow controllers. However, probes of link layer protocols use a specific packet header, which represents a single point in the flow space, and is not applicable to all the slices in the network unless it is handled in a different way by the FSM. So, to overcome this challenge, we propose a stochastic probing technique to validate the given topology. In this technique, a point in the flowspace of link's tail end port is randomly picked and a corresponding packet header is pushed out through the tail end port. Connectivity between the two endpoints is verified by comparing the header of the packet received against the header that was sent out from the other end. In this way, the user can rule out any misconfiguration of intermediate equipment before the network goes live.

**CiruitSwitching module**: This module is responsible for computing shortest paths between every pair of end points in the network and assigning a unique PATH IDentifier (PATHID), when they become active for the first time. However, an additional header field is required to identify the direction of the flow -- we propose the use of destination MAC address field and replace it with a part of the destination switch's DPID (Data Path IDentifier). Instead of matching a set of packet header fields, the intermediate switches on a route, forward packets based on the PATHID. Edge switches are responsible for replacing the source and destination MAC address fields with the PATHID and DPID respectively,

Figure [3]: Transit controller architecture

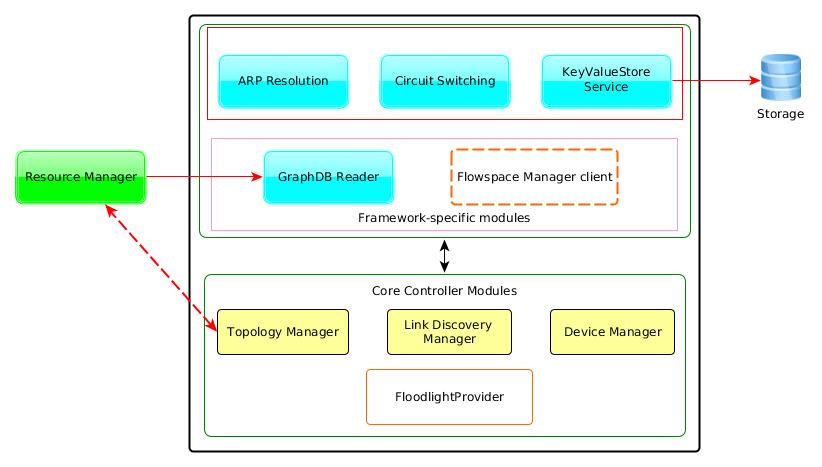
and vice-verse. We chose source MAC address field to replace with PATHID, because parsing of the packet beyond Layer 2 is computationally expensive in commodity hardware. One added advantage of this approach is, the number of active flow entries in the intermediate switches are drastically reduced, as all the traffic between two edge switches is forwarded based on a unique identifier. Moreover, this method doesn't require an additional header field, unlike the Multiprotocol Label Switching (MPLS) mechanism. Thus, improving the scalability of the network. Since PATHID replaces source MAC address field, the module ensures that the resulting packet header confines to the flowspace of the interfaces along the path.

**ARP Resolution module:** Today's OpenFlow networks suffer from repetitive flooding of ARP packets, as the default action taken by the controller to handle a packet bound to an unknown host is to broadcast it. The cost of ARP broadcast storms is minimal in small-scale networks, but it is not negligible in large-scale multi-domain networks, especially the increase in traffic on the control channel between the controller and the switches. This module ensures that ARP requests are resolved in a timely manner with minimal broadcast of packets. In essence, it keeps track of the hosts' IP to MAC address mapping -- we call it ARP table -- and creates proper responses to new ARP requests for known hosts. Moreover, it absorbs the ARP reply from an end host at the edge switch, and forwards it to the device that initiated the ARP request. In this way, it reduces ARP broadcast storms in the network.

**KeyValueStore Service module:** ﻿It is a simple module introduced into the controller to push the in-memory state of the controller to an offline storage system for future retrieval, in case of any unexpected failure or error. Firstly, we implemented a generic interface to a key value store so that users can plug in any storage system. We specifically developed a module extending the generic interface to store the state of the controller in Voldemort [25], a distributed key value storage system. We chose Voldemort for two reasons, its superior write performance and simple interface to the distributed storage system. This module exports a plain interface to other modules, enabling them to create new stores (tables) and save module-specific information in the form of key-value pairs. As of now, we push the topology at periodic intervals, while PATHIDs and the ARP table are updated in the system whenever new entries are created or existing ones are modified. This ensures that the controller is fault-tolerant -- a standby controller can take over immediately by restoring the state in the offline storage.

**4.3 Domain Controller**

The architecture of the domain controller is similar to that of the transit controller except for the inclusion of Link Discover Manager, which replaces the Topology Validation module. Since the domain controller owns the complete flowspace, the Link Discovery Manager uses LLDP packets to discover the topology. Another notable difference is, when the graph DB Reader module retrieves details about the delegated resources, it flags them as unavailable to the domain controller and inserts appropriate rules into the FSM through the interface exported by the Flowspace manager client (FSMC) module. Thus, authorizing the delegation of resources to the transit controller. For example, consider the figure [2], RM of domain 'A' delegates the source IP address space, 192.168.1.0/24, on all the interfaces to a transit controller. The GraphDB reader module on going through the graph database, generates rules similar to "delegate flow space (\*, 192.168.1.0/24, 192.168.1.0/24, \*), switch DPID, port ID to a transit controller <X> " and passes them onto the FSMC module, which then installs rules into the FSM in an appropriate format supported by the FSM. In addition, GraphDB module, modifies the flowspace of the domain controller and the corresponding rules in FSM to reflect the changes in flowspace.

Figure [4]: Architecture of the domain controller 

The physical topology consists of two domains, each containing three switches controlled by individual domain controllers. And Flowvisor plays the role of FSM, in both the domains. All the switches are an instance of OpenVSwitch running inside a virtual machine allocated across servers of the ExoGENI infrastructure. OpenVSwitch is a software-based virtual Openﬂow switch. It implements the data plane in the kernel and the control plane as a user space process. Each instance of OpenVSwitch runs on a 2000 MHZ single core virtual machine with 500 MB of RAM and 4KB cache. The domain and the transit controllers run on similar virtual machines, and are an instance of the modified Floodlight controller as described in sections 4.2 and 4.3. Lastly, each controller is attached to a Voldemort key-value system, which is hosted in a similar virtual machine to the controllers and the switches.

**5.1 Virtual topology, a subset of the physical topology**

In the first scenario, transit controller is delegated a part of the flowspace represented by destination mac address range from 02:00:00:00:00:00 to 02:00:00:00:00:1F. And source and destination IP address space, 192.168.99.0/24, on all the interfaces except for the ones connected to the hosts, in both the domains. The rest of the flowspace on all the interfaces is owned by the domain controller of respective domains, while the transit controller is delegated with entire flowspace on interfaces connecting the hosts [Host1-Host4] to the switches. Figure [6] shows the state of the domain controllers and the transit controller when flowvisors A & B are notified about the delegations -- as of now, it is manually done, but as stated earlier, we would like to automate it in the future. As you can see, the transit controller is connected to all the switches, but has no links interconnecting them, as it is not made aware of the interfaces and corresponding flowspaces on each switch yet. On the other hand, domain controllers learn about the topology of their network with the help of LLDP packets.

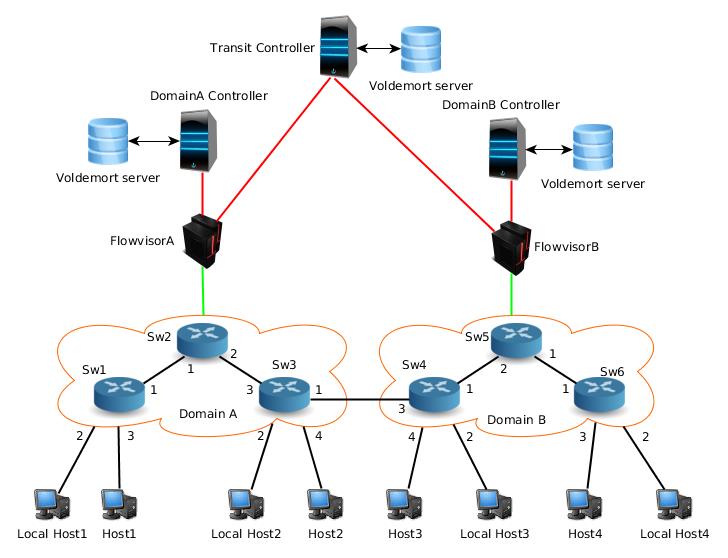
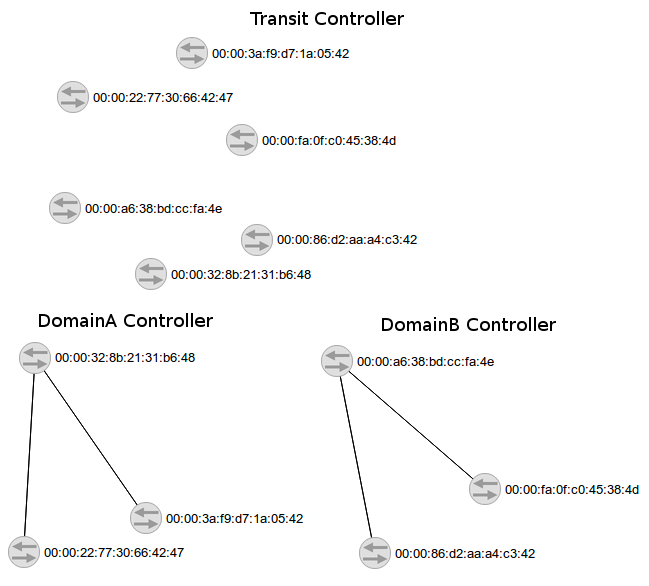


Figure [5]:Experimental setup

Now, when we manually emulate the behavior of RM, and notify the transit controller about the delegations in domain A, it starts the topology validation process as described in section 4.2. The state of the transit controller after it verifies the links, is shown below in figure [7]. As you can see, only the switches in domain A are interconnected while the rest of them are in isolation from each other. However, as soon as the transit controller receives resource delegations from domain B's RM, it validates the links between the switches and updates its view of the topology, which can be seen in figure [8]. With the exception of the link connecting the two domains, the topology of the transit controller's network appears similar to the actual topology. We bridge the gap between the two views of the network by informing the transit controller about the inter domain link and the flowspace allocated to it on the endpoints of the link, through a serialized DEX graph database.

As stated earlier, one important aspect of our framework's functionality is, multiple controllers operating in the same network without any conflicts. In order to prove that, we start two bidirectional flows between hosts connected to switch 1 and 3. We notice that there is no packet loss reported in either of the bidirectional streams. Furthermore, we verify the flow rules installed by both the controllers in the core and edge switches. From the figures [10] and [11], it is clear that both, transit as well as domain controller is using PATHID based forwarding, as the source and destination MAC addresses are swapped with PATHID (02:00:00:00:00:00/20) -- PATHIDs start with 02 instead of 00 because, we need to adhere to IEEE standards and ensure they are locally administered MAC addresses -- and destination switch DPID respectively, and vice-versa, at the edge switch. In the core switches, as expected, packets are matched based on the PATHID, the destination switch DPID and the input port.

Figure [6]: State of the domain and transit controllers at the start of the experiment

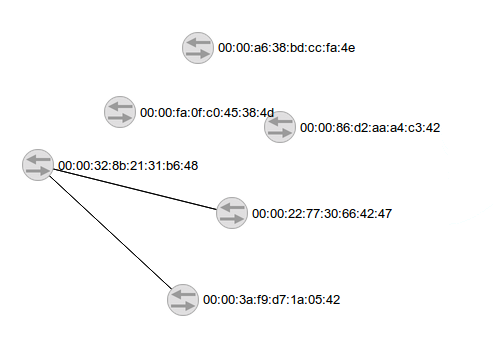


Figure [7]: State of the transit controller after it has verified the resource delegations in domain 'A'

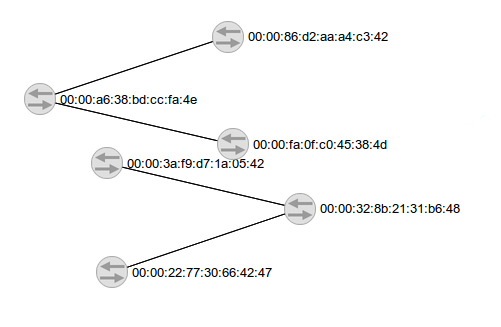


Figure [8]: State of the trasnit controller after it has verified the resource delegations in domain 'B'

**5.2 True Virtual Topology**

In this scenario, the transit controller is presented with a topology in which some of the switches are hidden and replaced with virtual links. Specifically, switch 5 is removed and replaced with a virtual link between switch 4 and 6. In addition, a flowspace mismatch -- dissimilar source and destination IP address space -- is introduced between the two end points of the virtual link. To simplify the experiment, mismatch is introduced only in one direction of the flow, i.e. in the direction towards switch 6. We believe this is a valid scenario in the real world, as over a period of time, flowspace may

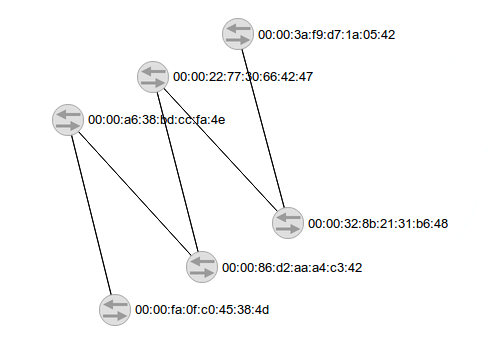


Figure [9]: State of the transit after it has verified all the resources delegated to it.

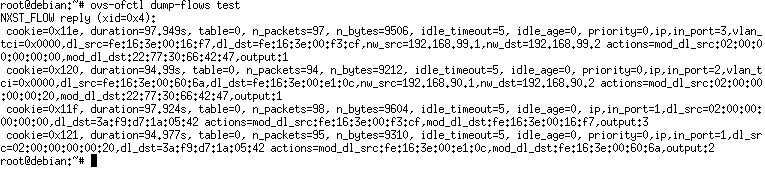
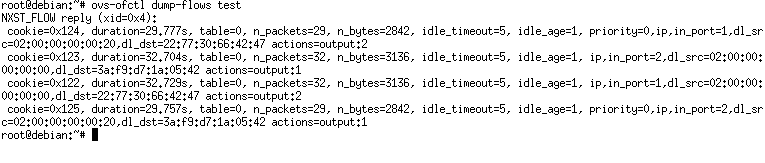


Figure [10]: Entries in the forwarding table of Switch 1

Figure [11]: Entries in forwarding table of switch 2

suffer from fragmentation and the RM may not be able to allocate similar flowspace on the two interfaces connected by the virtual link.

Figure [12] shows the state of the transit controller after it has validated the virtual topology given to it. As you can see, the domain controller is successful in hiding switch 5 from the transit controller as switch 4 and 6 appear to be directly connected to each other. To verify the virtual link status, we start a bidirectional flow between host 3 (192.168.99.3) and 4 (192.168.99.94). Figure [13] illustrates, the flow entries programmed by the transit controller in the forwarding table of switch 6. From the figure, it is clear that the transit controller, being aware of the non-empty "Rules" attribute of the virtual link, expects packets with different source and destination IP addresses compared to the actual ones, and reverts them back to the original values. Thus, ensuring full connectivity between the hosts. However, for the traffic to flow uninterruptedly between the hosts, the domain controller/FSM has to install appropriate flow rules in the intermediate switches -- switch 5 in this case, -- which is not the case here, as it is done manually. But, we plan to build a superior flowvisor with all the missing functionality to overcome this challenge.

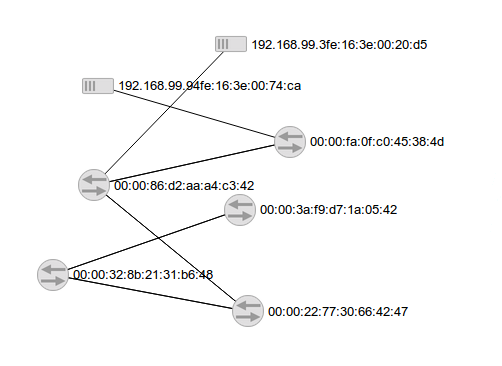
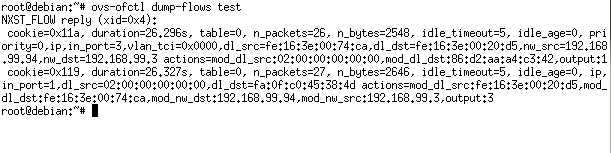


Figure [12]: State of the transit controller after it has verified the resource delegations in both the domains.

Figure [13]: Entries in the forwarding table of switch 6.

**5.3 Performance Evaluation**

In this subsection, we evaluate the performance of our topology verification process -- the stochastic probing method described in section [4.2] -- against the traditional Link Layer based topology discovery method, LLDP. We modified the LinkDiscovery module in Floodlight controller to ensure we only measure the time taken by the controller to send and receive LLDP packets. We performed the experiment on a test bed built on top of ExoGENI infrastructure. The topology of the test bed is as shown in the below figure [14]. We emulate a network of Openflow switches by running OpenVSwitch in virtual machines allocated by ExoGENI across different geographic locations. Open vSwitch is a software-based virtual Openﬂow switch. It implements the data plane in kernel and the control plane as a user space process. Each instance of OpenVSwitch runs on a 2000 MHZ single core virtual machine with 500 MB of RAM and 4KB cache. The latency on the links connecting the switches is 50 us. All the switches connect to a Flowvisor, which acts as a proxy of the Floodlight controller running on a similar virtual machine to that of the switches and itself. Although Flowvisor is not necessary in this scenario, we included it in our experiments to simulate the real world scenario of multi-domain networks, where each domain hosts a Flowvisor or anything that offers similar functionality.

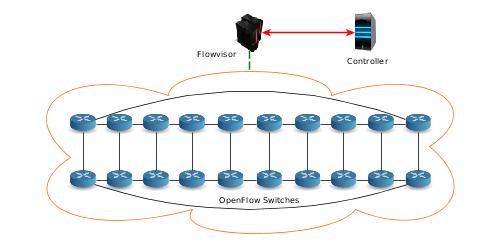


Figure [14]: Experimental setup to evaluate the performance of the proposed delegation framework.

We evaluated the performance of the two methods in various topologies with N nodes and L links. And present the average, maximum and minimum times for each method, as shown in figures [15, 16, 17]. We also quantify the time taken by the controller to process the delegations represented in the form of a serialized DEX graph. LLDP method takes the shortest time in all the three cases, while our method is marginally slower, as it randomly samples the flowspace to generate probes unlike the LLDP method, which uses a standard packet header. The processing of the GraphML-based description of delegated resources dominates the entire process. However, we believe that, performance can be improved by using more efficient forms of representations. For instance, if the ingress and egress flowspace of a port are identical, which is the case most of the times, then instead of using two unidirectional links, we can represent the connection between two ports with a single bi-directional link. Thus, reducing the size and time to process the graph.

**6. Conclusion**

In this paper, we have presented the overall design of our delegation framework and specific details about the two types of OpenFlow controllers deployed in the proposed architecture. In addition, we also showed that the new method to discover topology is marginally slower compared to LLDP, a traditional Link Layer protocol based topology discovery method. We conclude by presenting some of the use cases that we believe are critical for the deployment of our proposed framework in software-defined networks:

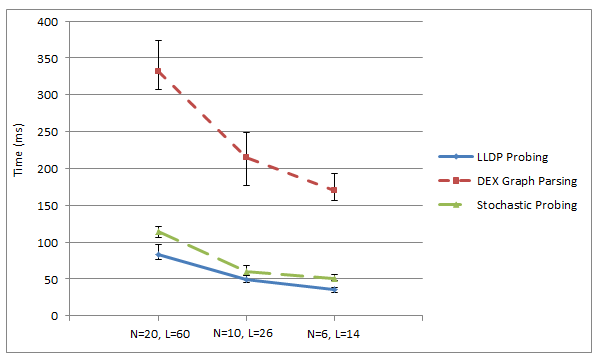


Figure [15]: Shows the comparison between times taken by probing methods.

1. **Efficient usage of flow space:** After provisioning multiple tenants with different flowspace and reclaiming expired delegations over a period of time, network providers will face the problem of flowspace fragmentation. By explicitly communicating the delegated flowspace on every link, and allowing for label translation in some of the delegated nodes, if not all of them, and as an attribute of the delegated links, proposed framework allows for employment of efficient policies to allocate labels to users in such scenarios.
2. **Dynamic resource allocation:** Resource usage of tenants is not constant all the time, and they vary considerably. For instance, consider the case of an e-commerce client, whose traffic varies with time of the year, requiring a different amount of bandwidth to handle the traffic. With the FSM and RM modules in place, our framework enables the provider to dynamically allocate additional bandwidth for a short period of time, thus optimizing the allocation of resources.
3. **Efficient resource utilization:** With explicit communication of delegated resources in a finer detail, our framework allows user controllers to switch traffic efficiently while optimizing resource usage. For example, consider the network in the below figure [16]. A user is allocated a bandwidth of 1 Gbps on all the links except for BD and AE, restricting him to a rate of 100 Mbps. Now, user needs to switch a traffic stream between hosts connected to Switches C and D. A standard controller not aware of bandwidth constraints on link BD will switch the traffic over the shortest path (C→ B → D). Thus, resulting in bandwidth throttling on link BD. However, in our framework, the tenant controller, which is aware of the bandwidth constraints, can adapt its switching policy and load balance the traffic over the two paths.

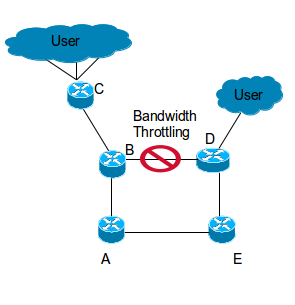


Figure [16]: A typical use case scenario of our framework

1. **Ability to introduce new revenue sources:** By clearly defining the resources delegated to a user in detail, and ensuring that the user cannot access any additional resources, the provider, apart from the usual methods [26], can now charge tenants based on class of hardware resources delegated to them. For instance, providers can categorize users based on the amount of buffer space allocated on switching devices or access to VLAN translation capability inside the switch.

As part of our future work, we plan to add a new module to the domain controller that programs the switches hidden by virtual links, so that traffic flows uninterrupted over them. We also plan to implement an interface between the domain controller and Resource manager as well as one between an advanced FlowVisor -- consists of an additional module called authorization framework, which enables recursive delegation of resources in a secure way -- and RM.

**References**

1. N. Leavitt, “Is cloud computing really ready for prime time?” Computer, vol. 42, no. 1, pp. 15 –20, jan. 2009
2. GENI, Global Environment for Network Innovations project, http://www.geni.net.
3. OFELIA project, http://www.fp7-ofelia.eu.
4. R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McK- eown, and G. Parulkar, “Flowvisor: A network virtualization layer,” OpenFlow Switch Consortium, Tech. Rep, 2009.
5. E. Salvadori, R. D. Corin, A. Broglio, and M. Gerola, “Generalizing virtual network topologies in openflow-based networks,” in Global Telecommunications Conference (GLOBECOM 2011), 2011 IEEE. IEEE, 2011, pp. 1–6.
6. R. D. Corin, M. Gerola, R. Riggio, F. De Pellegrini, and E. Salvadori, “Vertigo: Network virtualization and beyond,” in Software Defined Networking (EWSDN), 2012 European Workshop on. IEEE, 2012, pp. 24–29.
7. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “OpenFlow: enabling innovation in campus networks,” ACM SIGCOMM Computer Communication Review, 38 (2):69–74, April 2008.
8. Sushant Jain , Alok Kumar , Subhasree Mandal , Joon Ong , Leon Poutievski , Arjun Singh , Subbaiah Venkata , Jim Wanderer , Junlan Zhou , Min Zhu , Jon Zolla , Urs Hölzle , Stephen Stuart , Amin Vahdat, “B4: experience with a globally-deployed software defined wan,” Proceedings of the ACM SIGCOMM 2013 conference on SIGCOMM, August 12-16, 2013, Hong Kong, China.
9. Brandon Heller , Rob Sherwood , Nick McKeown, “The controller placement problem,” Proceedings of the first workshop on Hot topics in software defined networks, August 13-13, 2012, Helsinki, Finland.
10. K. Phemius, M. Bouet, J. Leguay, “DISCO : Distributed Multi-domain SDN Controllers,” http://arxiv.org/abs/1308.6138v2.
11. A. Sadasivarao, A. Lake, C. Liou, S. Syed, P. Pan, C. Guok, and I. Monga, “Open Transport Switch - A Software Defined Networking Architecture for Transport Networks,” HotSDN’13, pp. 115–120, 2013.
12. J. McCauley, A. Panda, M. Casado, T. Koponen, S. Shenker, and U. C. Berkeley, “Extending SDN to Large-Scale Networks,” Open Networking Summit, pp. 1–2, 2013.
13. Yaping Zhu , Rui Zhang-Shen , Sampath Rangarajan , Jennifer Rexford, “Cabernet: connectivity architecture for better network services,” Proceedings of the 2008 ACM CoNEXT Conference, p.1-6, December 09-12, 2008, Madrid, Spain.
14. N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras, “Dex: A high-performance graph database management system,” in Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on. IEEE, 2011, pp. 124–127.
15. Pan, P., Ed., Swallow, G., Ed., and A. Atlas, Ed., "Fast Reroute Extensions to RSVP-TE for LSP Tunnels," RFC 4090, May 2005.
16. Peyman Kazemian , George Varghese , Nick McKeown, “Header space analysis: static checking for networks,” Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, April 25-27, 2012, San Jose, CA.
17. Drutskoy, D., Keller, E., Rexford, J., "Scalable Network Virtualization in Software-Defined Networks," IEEE Internet Computing, vol. 17, no. 2, pp. 20-27, March-April, 2013.
18. N. Chowdhury, M. Rahman, and R. Boutaba, “Virtual network embedding with coordinated node and link mapping,” in IEEE INFOCOM, pp. 783–791, April 2009.
19. M. Nascimento and C. R. Et al., “Virtual routers as a service: the routeflow approach leveraging software-defined networks,” in Proc. Of ACM CFI, Seoul, Korea, 13-15 June 2011, pp. 34–37.
20. “Floodlight OpenFlow Controller.” [Online]. Available: http://floodlight.openflowhub.org
21. “RabbitMQ.” [Online]. Available: http://www.rabbitmq.com
22. “AMQP.” [Online]. Available: http://www.amqp.org
23. M. S. Malik, M. Montanari, J. H. Huh, R. B. Bobba, and R. H. Camp- bell, “Towards sdn enabled network control delegation in clouds,” in Dependable Systems and Networks (DSN), 2013 43rd Annual IEEE/IFIP International Conference on. IEEE, 2013, pp. 1–6.
24. "POX OpenFlow Controller." [Online]. Available: <https://openflow.stanford.edu/display/ONL/POX+Wiki>
25. “Project Voldemort.” [Online]. Available: <http://www.project-voldemort.com/voldemort/>
26. C. Courcoubetis, Frank P. Kelly, Vasilios A. Siris and Richard Weber, "A study of simple usage-based charging schemes for broadband networks," Proc. 4th Int\'l. Conf. Broadband Commun., pp.209 -221 1998
27. G. Popek and R. Goldberg "Formal requirements for virtualizable third generation architectures", *Commun. ACM*, vol. 17, pp.412 -421 1974

1. Borrowing from the GENI literature, we call an instance of a virtual network a slice, and two distinct virtual networks on the same physical hardware slices [↑](#footnote-ref-2)
2. This work includes major contributions from Dr. Ilya Baldin and Dr. Shu Huang, and it is included here so that the reader can appreciate rest of the work. [↑](#footnote-ref-3)