

ShadowNet: A Platform for Rapid and Safe Network Evolution

Xu Chen Z. Morley Mao Jacobus Van der Merwe
University of Michigan AT&T Labs - Research

Abstract

The ability to rapidly deploy new network services, service features, and operational tools, without impacting existing services, is a significant challenge for all service providers. In this paper we address this problem by the introduction of a platform called ShadowNet. ShadowNet exploits the strong separation provided by modern compute and network equipment between logical functionality and the physical infrastructure. It allows logical topologies of compute servers, networking equipment and links to be dynamically created, then instantiated to and managed on the physical infrastructure. ShadowNet is a sharable, programmable, and composable infrastructure, consisting of carrier-grade equipment. Furthermore, it is a fully operational network that is connected to, but functionally separate from the provider production network. By exploiting the strong separation support, ShadowNet allows multiple technology and service trials to be executed in parallel in a realistic operational setting, without impacting the production network. In this paper, we describe the ShadowNet architecture and the control framework designed for its operation and illustrate the utility of the platform. We present our prototype implementation and demonstrate the effectiveness of the platform through extensive evaluation.

1 Introduction

Effecting network change is fundamentally difficult. This is primarily due to the fact that modern networks are inherently shared and multi-service in nature, and any change to the network has the potential to negatively impact existing users and services. Historically, production quality network equipment has also been proprietary and closed in nature, thus further raising the bar to the introduction of any new network functionality. The negative impact of this state of affairs has been widely recognized as impeding innovation and evolution [22]. Indeed at a macro-level, the status quo has led to calls for a clean slate redesign of the Internet which in turn has produced efforts such as GENI [3] and FEDERICA [2].

In the work presented in this paper we recognize that at a more modest micro-level, the same fundamental problem, *i.e.*, the fact that network change is inherently difficult, is a major *operational* concern for service providers. Specifically, the introduction of a new service

or service features typically involve long deployment cycles: Configuration changes to network equipment is meticulously lab-tested before staged deployments are performed in an attempt to reduce the potential of any negative impact on existing services. The same applies to any new tools to be deployed in support of network management and operations. This is especially true as network management tools are evolving to be more sophisticated and capable of controlling network functions in an automated closed-loop fashion [24, 9, 7]. The operation of such tools depends on the actual state of the network, presenting further difficulties in terms of testing in a lab environment due to difficulties to artificially recreate realistic network conditions in a lab setting.

In this paper we address these concerns through a platform called *ShadowNet*. ShadowNet is designed to be an operational trial/test network consisting of ShadowNet *nodes* distributed throughout the backbone of a tier-1 provider in the continental U.S. Each ShadowNet node is composed of a collection of carrier-grade equipment, namely routers, switches and servers. Each node is connected to the Internet as well as to other ShadowNet nodes via a (virtual) backbone.

ShadowNet provides a *sharable, programmable and composable* infrastructure to enable the rapid trialing or deployment of new network services or service features, or evaluation of new network management tools in a realistic operational network environment. Specifically, via the Internet connectivity of each ShadowNet node, traffic from arbitrary end-points can reach ShadowNet. ShadowNet connects to and interacts with the provider backbone much like a customer network would, and as such the “regular” provider backbone, just like it would protect itself from any other customers, is isolated from the testing and experimentation that take place within ShadowNet. In the first instance, ShadowNet provides the means for testing services and procedures for subsequent deployment in a (separate) production network. However, in time we anticipate ShadowNet-like functionality to be provided by the production network itself to directly enable rapid but safe service deployment.

ShadowNet has much in common with other test networks [10, 26, 21]: (i) ShadowNet utilizes “virtualization” and/or “partitioning” capabilities of equipment to enable *sharing* of the platform between different concurrently running trials/experiments. (ii) Equipment

in ShadowNet nodes are *programmable* to enable experimentation and the introduction of new functionality. (iii) ShadowNet allows the dynamic *composition* of test/trial topologies.

What makes ShadowNet unique, however, is that this functionality is provided in an *operational network on carrier-grade equipment*. This is critically important for our objective to provide a rapid service deployment/evaluation platform, as technology or service trials performed in ShadowNet should mimic technology used in the provider network as closely as possible. This is made possible by recent vendor capabilities that allow the partitioning of physical routers into subsets of resources that essentially provide logically separate (smaller) versions of the physical router [16].

In this paper, we describe the ShadowNet architecture and specifically the ShadowNet control framework. A distinctive aspect of the control framework is that it provides a clean separation between the *physical-level* equipment in the testbed and the *user-level* slice specifications that can be constructed “within” this physical platform. A *slice*, which encapsulates a service trial, is essentially a container of the service design including devices and connectivity specification and placement. Once instantiated, a slice also contains the allocated physical resources to the service trial. Despite this clean separation, the partitioning capabilities of the underlying hardware allows virtualized equipment to be largely indistinguishable from their physical counterparts, except that they contain fewer resources. The ShadowNet control framework provides a set of interfaces allowing users to programatically interact with the platform to manage and manipulate their slices.

We make the following contributions in this work:

- Present a sharable, programmable, and composable network architecture which employs strong separation between user-level topologies/slices and their physical realization (§2).
- Present a network control framework that allows users to manipulate their slices and/or the physical resource contained therein with a simple interface (§3).
- Describe physical-level realizations of user-level slice specifications using carrier-grade equipment and network services/capabilities (§4).
- Present a prototype implementation (§5) and evaluation of our architecture (§6).

2 ShadowNet overview

In this paper, we present ShadowNet which serves as a platform for rapid and safe network change. The primary goal of ShadowNet is to allow the rapid composition of distributed computing and networking resources, contained in a *slice*, realized in carrier-grade facilities

which can be utilized to introduce and/or test new services or network management tools. The ShadowNet control framework allows the network-wide resources that make up each slice to be managed either collectively or individually.

In the first instance, ShadowNet will limit new services to the set of resources allocated for that purpose, *i.e.*, contained in a slice. This would be a sufficient solution for testing and trying out new services in a realistic environment before introducing such services into a production network. Indeed our current deployment plans espouse this approach with ShadowNet as a separate overlay facility [23] connected to a tier-1 production network. Longer term, however, we expect the base functionality provided by ShadowNet to evolve into the production network and to allow resources and functionality from different slices to be gracefully merged under the control of the ShadowNet control framework.

In the remainder of this section we first elaborate on the challenges network service providers face in effecting network change. We describe the ShadowNet architecture and show how it can be used to realize a sophisticated service. Several experimental network platforms are compared against it, showing only ShadowNet really meets the requirements for realistic network testing. Finally we describe the architecture of the primary system component, namely the ShadowNet controller.

2.1 Dealing with network change

There are primarily three drivers for change in modern service provider networks:

Growth demands: Fueled by an increase in broadband subscribers and media rich content, traffic volumes on the Internet continue to show double digit growth rates year after year. The implication of this is that service providers are required to increase link and/or equipment capacities on a regular basis, even if the network functionality essentially stays the same.

New services and technologies: Satisfying customer needs through new service offerings are essential to the survival of any network provider. “Service” here spans the range from application-level services like VoIP and IPTV, connectivity services like VPNs and IPv4/IPv6 transport, traffic management services like DDoS mitigation or content distribution networks (CDNs), or more mundane (but equally important and complicated) service features like the ability to signal routing preferences to the provider or load balancing features.

New operational tools and procedures: Increasing use of IP networks for business critical applications is leading to growing demands on operational procedures. For example, end-user applications are often very intolerant of even the smallest network disruption leading to the

deployment of methods to decrease routing convergence in the event of network failures. Similarly, availability expectations, in turn driven by higher level business needs, make regularly planned maintenance events problematic, leading to the development of sophisticated operational methods to limit their impact.

As we have alluded to already, the main concern of **any** network change is that it might have an impact on existing network services, because networks are inherently shared with known and potentially unknown dependencies between components. An example would be the multi-protocol extensions to BGP to enable MPLS-VPNs or indeed any new protocol family. The change associated with rolling out a new extended BGP stack clearly has the potential to impact existing IPv4 BGP interactions, as bugs in new BGP software could negatively impact the BGP stack as a whole.

Note also that network services and service features are normally “cumulative” in the sense that once deployed and used, network services are very rarely “switched off”. This means that over time the dependencies and the potential for negative impact only increase rather than diminish.

A related complication associated with any network change, especially for new services and service features, is the requirement for corresponding changes to a variety of operational support systems including: (i) configuration management systems (new services need to be configured typically across many network elements), (ii) network management systems (network elements and protocols need to be monitored and maintained), (iii) service monitoring systems (for example to ensure that network-wide service level agreements, *e.g.*, loss, delay, or video quality, are met), (iv) provisioning systems (*e.g.*, to ensure the timely build-out of popular services). ShadowNet does not address these concerns per se. However, as described above, new operational solutions are increasingly more sophisticated and automated, and ShadowNet provides the means for safely testing out such functionality in a realistic environment.

Our ultimate goal with the ShadowNet work is to develop mechanisms and network management primitives that would allow new services and operational tools to be safely deployed directly in production networks. However, as we describe next, in the work presented here we take the more modest first step of allowing such actions to be performed in an operational network that is separate from the production network, which is an important transition step.

2.2 ShadowNet architecture

Different viewpoints of the ShadowNet network architecture are shown in Figures 1(a) and (b). Figure 1(a) shows the topology from the viewpoint of the tier-1

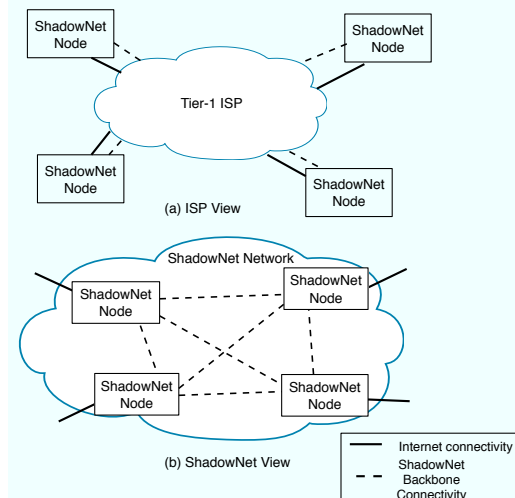


Figure 1: ShadowNet network viewpoints

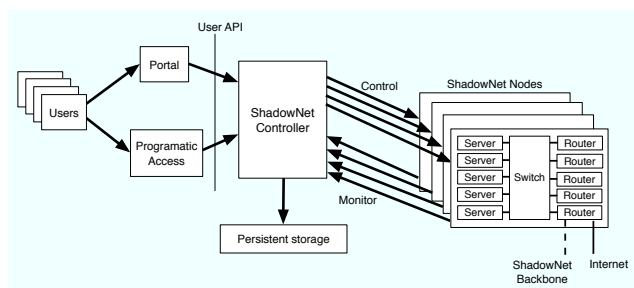


Figure 2: ShadowNet functional architecture

provider. ShadowNet nodes connect to the provider network, but is essentially separated from it. Each ShadowNet node has connectivity to other ShadowNet nodes as well as connectivity to the Internet. As shown in Figure 1(b), connectivity to other ShadowNet nodes effectively creates an overlay network [23] to form a virtual backbone among the nodes. Via the provided Internet connectivity, the ShadowNet address space is advertised (*e.g.*, using BGP) first to the provider network and then to the rest of the Internet. Thus ShadowNet effectively becomes a small provider network itself (*i.e.*, a *shadow* of the provider network).

The ShadowNet functional architecture is shown in Figure 2. Each ShadowNet node contains different types of computing and networking devices, such as servers, routers, and switches. Combined with the network connectivity received from the ISP, they complete the physical resource for ShadowNet. ShadowNet manages the physical resources and enables its users to share it. The devices provide virtualization/partitioning capabilities so that multiple logical devices can share the same underlying physical resource. For example, modern routers allow router resources to be partitioned so that several logical routers can be configured to run simultaneously and separately on a single physical router [16]. (Note

that modern routers are also programmable in both control and data planes [18].) Logical interfaces can be multiplexed from one physical interface via configuration and then assigned to different logical routers. We also take advantage of virtual machine technology to manage server resources [5]. This technology enables multiple operating systems to run simultaneously on the same physical machine and is already heavily used in cloud computing and data-center environments. To facilitate sharing connectivity, the physical devices in each ShadowNet node are connected via a configurable switching layer, which shares the local connectivity, for example using VLAN. The carrier-supporting-carrier capabilities enabled by MPLS virtual private networks (VPNs) [11, 15] offer strong isolation and are therefore an ideal choice to create the ShadowNet backbone.

As depicted in Figure 2, central to ShadowNet functionality is the *ShadowNet Controller*. The controller facilitates the specification and instantiation of a service trial in the form of a *slice* owned by a user. It provides a programmatic application programming interface (API) to ShadowNet users, allowing them to create the topological setup of the intended service trial or deployment. Alternatively users can access ShadowNet through a Web-based portal, which in turn will interact with the ShadowNet Controller via the user-level API. The ShadowNet Controller keeps track of the physical devices that make up each ShadowNet node by constantly monitoring them, and further manages and manipulates those physical devices to realize the user-level APIs, while maintaining a clean separation between the abstracted slice specifications and the way they are realized on the physical equipments. The user-level APIs also enable users to dynamically interact with and manage the physical instantiation of their slices. Specifically, users can directly access and configure each instantiated logical device.

ShadowNet allows a user to deactivate individual devices in a slice, or the slice as a whole, by releasing the allocated physical resources. ShadowNet decouples the persistent state from the instantiated physical devices, so that the state change associated with a device in the specification is maintained even if the physical instantiation is released. Subsequently, that device in the specification can be re-instantiated assuming sufficient resources, the saved state restored and thus the user perceived slice remains intact. For example, the configuration change made by the user to a logical router can be maintained and applied to a new instantiated logical router, even if the physical placement of that logical device is different.

2.3 Using ShadowNet

In this section we briefly describe an example usage scenario that illustrates the type of sophisticated network

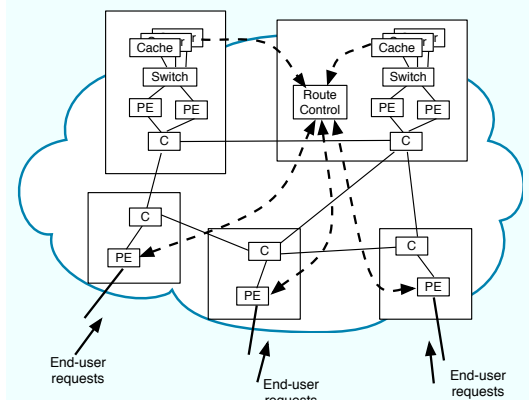


Figure 3: Usage scenario: load-aware anycast CDN.

service that can be tested using the ShadowNet infrastructure. We discuss the requirements and compare the difference among several experimental networking environments.

Assume that ShadowNet is to be used to run a customer trial of a *load-aware anycast content distribution network (CDN)* [9]. Figure 3 depicts how all the components of such a CDN can be realized on the ShadowNet platform. Specifically, a network, complete with provider edge (PE) and core (C) routers, can be dynamically instantiated to represent a small backbone network. Further, servers in a subset of the ShadowNet nodes can be allocated and configured to serve as content caches. A load-aware anycast CDN utilizes route control to inform BGP selection based on the cache load, *i.e.*, using BGP, traffic can be steered away from overloaded cache servers. In ShadowNet this BGP speaking route control entity can be instantiated on either a server or a router depending on the implementation. Appropriate configuration/implementation of BGP, flow-sampling, and server load monitoring complete the infrastructure picture. Finally, actual end-user requests can be directed to this infrastructure, *e.g.*, by resolving a content URL to the anycast address(es) associated with and advertised by the CDN contained in the ShadowNet infrastructure.

Using this example we can identify several capabilities required of the ShadowNet infrastructure to enable realistic service evaluation (see Table 1): (i) To gain confidence in the equipment used in the trial it should be the same as, or similar to, equipment used in the production network (*production-grade devices*). (ii) To thoroughly test load feedback mechanisms and traffic steering algorithms, it requires participation of significant numbers of customers (*realistic workloads*). (iii) This in turn requires sufficient network capacity (*high capacity backbone*). (iv) Realistic network and CDN functionality require realistic network latencies and geographic distribution (*geographic coverage*). (v) Finally, the CDN control framework could dynamically adjust the resources

	SN	EL	PL	VN
Production grade devices	Y	N	N	N
Realistic workloads	Y	N	Y	Y
High capacity backbone	Y	N	N	Y
Geographical coverage	Y	N	Y	Y
Dynamic reconfiguration	Y	N	N	N

Table 1: Capability comparison between ShadowNet (SN), EmuLab (EL), PlanetLab (PL) and VINI (VN)

allocated to it based on the offered load (*dynamic reconfiguration*).

To illustrate the utility of the ShadowNet infrastructure, Table 1 also shows comparable capabilities provided by other experimental testbeds. We describe these facilities next.

Emulab achieves flexible network topology through emulation within a central testbed environment. There is a significant gap between emulation environments and real production networks. For example, software routers typically do not provide the same throughput as production routers with hardware support. As EmuLab is a closed environment, it is incapable of combining real Internet workload into experiments. Compared to EmuLab, the ShadowNet infrastructure is distributed, thus the resource placement in ShadowNet more closely resembles future deployment phases. In EmuLab, an experiment in a slice is allocated a fixed set of resources during its life cycle. ShadowNet, on the other hand, can change the specification *dynamically*. In the CDN example, machines for content caches and network links can be dynamically spawned or removed in response to increased or decreased client requests.

PlanetLab has been extremely successful in academic research, especially in distributed monitoring and P2P research. It achieves its goal of amazing geographical coverage, spanning nodes to all over the globe, obtaining great end-host visibility. The PlanetLab nodes, however, are mostly connected to educational networks without abundant upstream or downstream bandwidth. PlanetLab therefore lacks the capacity to realize a capable *backbone* between PlanetLab nodes. ShadowNet, on the other hand, is built upon a production ISP network, having its own virtual backbone with bandwidth and latency guarantees. This pushes the tested service closer to the core of the ISP network, where the actual production service would be deployed.

VINI is closely tied with PlanetLab, but utilizes Internet2 to provide a realistic backbone. Like EmuLab and PlanetLab, VINI runs software routers (XORP and Click), the forwarding capacity of which lags behind production devices. This is mostly because its focus is to use commodity hardware to evaluate new Internet archi-

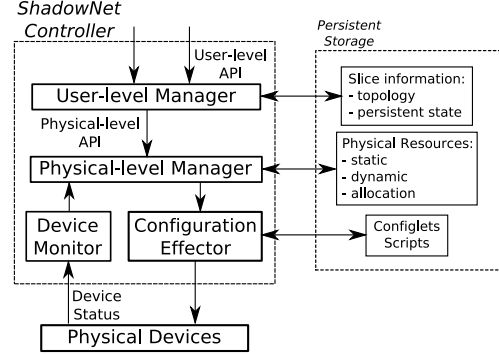


Figure 4: The ShadowNet controller

tures, which is different from the service deployment focus of ShadowNet. VINI and PlanetLab are based on the same control framework. Similar to EmuLab, it lacks the capability of changing slice configurations dynamically, *i.e.*, not closing the loop for more adaptive resource management, a functionality readily available in ShadowNet.

2.4 The ShadowNet Controller

The ShadowNet controller consists of a user-level manager, a physical-level manager, a configuration effector and a device monitor, as shown in Figure 4. We describe each component below. The current ShadowNet design utilizes a centralized controller that interacts with and controls all ShadowNet nodes.

2.4.1 User-level manager

The user-level manager is designed to take the input of user-level API calls. Each API call corresponds to an action that the users of ShadowNet are allowed to perform. A user can create a topological specification of a service trial (§3.1), instantiate the specification to physical resources (§3.2), interact with the allocated physical resources (§3.3), and deactivate the slice when the test finishes (§3.4). The topology specification of a slice is stored by the user-level manager in persistent storage, so that it can be retrieved, revived, and modified over time. The user-level manager also helps maintain and manage the saved persistent state from physical instantiations (§3.3). By retrieving saved states and applying them to physical instantiations, advanced features, like device duplication, can be enabled (§3.5).

The user-level manager is essentially a network service used to manipulate configurations of user experiments. We allow the user-level manager to be accessed from within the experiment, facilitating network control in a closed-loop fashion. In the example shown in Figure 3, the route control component in the experiment can dynamically add content caches when user demand is high by calling the user-level API to add more computing and networking resource via the user-level manager.

2.4.2 Physical-level manager

The physical-level manager fulfills requests from the user-level manager in the form of physical-level API calls by manipulating the physical resources in ShadowNet. To do this, it maintains three types of information: 1) “static” information, such as the devices in each ShadowNet node and their capabilities; 2) “dynamic” information, *e.g.*, the online status of all devices and whether any interface modules are not functioning; 3) “allocation” information, which is the up-to-date usage of the physical resources. Static information is changed when new devices are added or old devices are removed. Dynamic information is constantly updated by the device monitor. The three main functions of the physical-level manager is to configure physical devices to spawn virtualized *device slivers* (§4.1) for the instantiation of user-level devices (§4.1.1) and user-level connectivities (§4.1.2), to manage their states (§4.4) and to delete existing instantiated slivers. A *sliver* is a share of the physical resource, *e.g.*, a virtual machine or a sliced physical link. The physical-level manager handles requests, such as creating a VM, by figuring out the physical device to configure and how to configure it. The actual management actions are performed via the configuration effector module, which we describe next.

2.4.3 Configuration effector

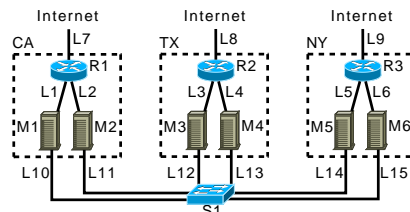
The configuration effector specializes in realizing configuration changes to physical devices. *Configlets* are parametrized configuration or script templates, saved in the persistent storage and retrieved on demand. To realize the physical-level API calls, the physical-level manager decides the appropriate configlet to use and generates parameters based on the request and the physical resource information. The configuration effector executes the configuration change on target physical devices.

2.4.4 Device monitor

A device monitor actively or passively determines the status of physical devices or components and propagates this “dynamic” information to the physical-level manager. Effectively, the device monitor detects any physical device failures in real time. As the physical-level manager receives the update, it can perform appropriate actions to mitigate the failure. The goal is to minimize any inconsistency of physical instantiation and user specifications. We detail the techniques in §4.5. Device or component recovery can be detected as well, and as such the recovered resource can again be considered usable by the physical-level manager.

3 Network service in a slice

A user of ShadowNet creates a service topology in the form of a *slice*, which is manipulated through the user-



```

$SL = AddUsrSlice();
$S1 = AddUsrSwitch($SL);
$R1 = AddUsrRouter($SL, "CA");
$M1 = AddUsrMachine($SL, "CA", "Debian");
$M2 = AddUsrMachine($SL, "CA", "Windows");
$L1 = AddUsrLink($M1, $R1); # similar for M2
$L10 = AddUsrLink($M1, $S1); # similar for M2
$L7 = AddToInternet($R1, "141.212.111.0/24");
# similar for "TX" and "NY"

```

Figure 5: Example of user-level API calls

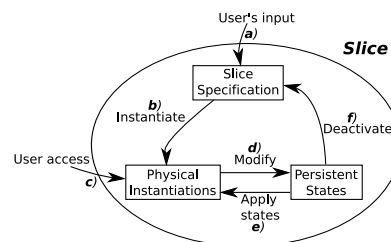


Figure 6: The slice life cycle

level API calls supported by the ShadowNet controller. The three layers embedded in a slice and the interactions among them are depicted in Figure 6 and detailed below. In this section, we outline the main user-exposed functionalities that the APIs implement.

3.1 Creating user-level specification

To create a new service trial, an authorized user of ShadowNet can create a *slice*. As a basic support, and usually the first step to create the service, the user specifies the topological setup through the user-level API (a in Figure 6). As an example, Figure 5 depicts the intended topology of a hypothetical slice and the API call sequence that creates it.

The slice created acts like a placeholder for a collection of *user-level objects*, including devices and connectivities. We support three generic types of user-level devices (UsrDevice): router (UsrRouter), machine (UsrMachine), and switch (UsrSwitch). Two UsrDevices can be connected to each other via a user-level link (UsrLink). User-level interfaces (UsrInt) can be added to a UsrDevice explicitly by the slice owner; however, in most cases, they are created implicitly when a UsrLink is added to connect two UsrDevices.

Functionally speaking, a UsrMachine (*e.g.*, *M1* in Figure 5) represents a generic computing resource, where the user can run service applications. A UsrRouter (*i.e.*, *R1*) can run routing protocols, filter and

forward packets, *etc.* Further, `UsrRouters` are programmable, allowing for custom router functionality. A `UsrLink` (*i.e.*, $L1$) ensures that when the `UsrDevice` on one end sends a packet, the `UsrDevice` on the other end will receive it. A `UsrSwitch` (*i.e.*, $S1$) provides a single broadcast domain to the `UsrDevices` connecting to it. `ShadowNet` provides the capability and flexibility of putting geographically dispersed devices on the same broadcast domain. For example, $M1$ to $M6$, although specified in different locations, are all connected to `UsrSwitch` $S1$. Besides internal connectivity among `UsrDevices`, `ShadowNet` can drive live Internet traffic to a service trial by allocating a public IP prefix for a `UsrInt` on a `UsrDevice`. For example, $L7$ is used to connect $R1$ to the Internet, allocating an IP prefix of $141.212.111.0/24$.

Besides creating devices and links, a user of `ShadowNet` can also associate properties with different objects, *e.g.*, the OS image of a `UsrMachine` and the IP addresses of the two interfaces on each side of a `UsrLink`. As a distributed infrastructure, `ShadowNet` allows users to specify location preference for each device as well, *e.g.*, California for $M1$, $M2$ and $R1$. This location information is used by the physical layer manager when placement is performed.

3.2 Instantiation

A user can instantiate some or all the objects in her slice onto physical resources (b in Figure 6). From this point on, the slice not only contains abstracted specification, but has associated physical resources that the instantiated objects in the specification are mapped to.

`ShadowNet` provides two types of instantiation strategies. First, a user can design a full specification for the slice and instantiate all the objects in the specification together. This is similar to what `Emulab` and `VINI` provide. As a second option, user-level objects in the specification can be instantiated upon request at any time. For example, they can be instantiated on-the-fly as they are added to the service specification. This is useful for users who would like to build a slice interactively and/or modify it over time, *e.g.*, extend the slice resources based on increased demand.

3.3 Device access & persistent slice state

`ShadowNet` allows a user to access the physical instantiation of the `UsrDevices` and `UsrLinks` in her slice, *e.g.*, logging into a router or tapping into a link (c in Figure 6). This support is necessary for many reasons. First, the user needs to install software on `UsrMachines` or `UsrRouters` and/or configure `UsrRouters` for forwarding and filtering packets. Secondly, purely from an operational point of view, operators usually desire direct access to the devices (*e.g.*, a terminal window on a server, or com-

mand line access to a router), rather than using a restricted set of function calls.

For `UsrMachines` and `UsrRouters`, we allow users to log into the device and make any changes they want (§4.3). For `UsrLinks` and `UsrSwitches`, we provide packet dump feeds upon request (§4.3). This support is crucial for service testing, debugging and optimization, since it gives the capability and flexibility of sniffing packets at any place within the service deployment without installing additional software on end-points.

Enabling device access also grants users the ability to change the persistent state of the physical instantiations, such as files installed on disks and configuration changes on routers. In `ShadowNet`, we decouple the persistent states from the physical instantiation. When the physical instantiation is modified, the changed state also become part of the slice (d in Figure 6).

3.4 Deactivation

The instantiated user-level objects in the specification of a slice can be deactivated, releasing the physical instantiations of the objects from the slice by giving it back to the `ShadowNet` infrastructure. For example, a user can choose to deactivate an under-utilized slice as a whole, so that other users can test their slices when the physical resources are scarce. While releasing the physical resource, we make sure the persistent state is extracted and stored as part of the slice (f in Figure 6). As a result, when the user decides to revive a whole slice or an object in the slice, new physical resources will be acquired and the stored states associated with the object(s) applied to it (e in Figure 6). Operationally speaking, this enables a user to deactivate a slice and reactivate it later, most likely on a different set of resources but still functioning like before.

3.5 Management support

Abstracting the persistent state from the physical instantiation enables other useful primitives in the context of service deployment. If we instantiate a new `UsrDevice` and apply the state of an existing `UsrDevice` to it, we effectively duplicate the existing `UsrDevice`. For example, a user may instantiate a new `UsrMachine` with only the basic OS setup, log into the machine to install necessary application code and configure the OS. With the support provided by `ShadowNet`, she can spawn several new `UsrMachines` and apply the state of the first machine. This eases the task of creating a cluster of devices serving similar purposes. From the `ShadowNet` control aspect, this separation allows sophisticated techniques to hide physical device failures. For example, a physical router experiences a power failure, while it hosts many logical routers as the instantiation of `UsrRouters`. In this case, we only need to create new instantiations on other avail-

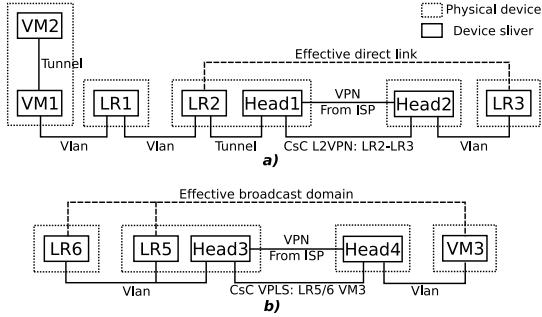


Figure 7: Network connectivity options.

able devices of the same type, and then apply the states to them. During the whole process, the slice specification, which is what the user perceives, is intact. Naturally, the slice will experience some downtime as a result of the failure.

4 Physical layer operations

While conceptually similar to several existing systems [10, 26], engineering ShadowNet is challenging due to the strong isolation concept it rests on, the production-grade qualities it provides and the distributed nature of its realization. We describe the key methods used to realize ShadowNet.

4.1 Instantiating slice specifications

The slice specification instantiation is performed by the ShadowNet controller in a fully automated fashion. The methods to instantiate on two types of resource are described as follows.

4.1.1 User-level routers and machines

ShadowNet currently utilizes VirtualBox [5] from Sun Microsystems, and Logical Routers [16] from Juniper Networks to realize *UsrMachines* and *UsrRouters* respectively. To instantiate a *UsrRouter* or a *UsrMachine*, a ShadowNet node is chosen based on the location property specified. Then all matching physical devices on that node are enumerated for availability checking, *e.g.*, whether a Juniper router is capable of spawning a new logical router. When there are multiple choices, we distribute the usage across devices in a round-robin fashion. Location preference may be unspecified because the user does not care about where the *UsrDevice* is instantiated, *e.g.*, when testing a router configuration option. In this case, we greedily choose the ShadowNet node where that type of device is the least utilized. The resource allocation problem in ShadowNet is largely determined by availability in each node and thus handled in a straightforward manner. When no available resource can be allocated, an error is returned.

4.1.2 User-level connectivity

The production network associated with ShadowNet provides both Internet connection and virtual backbone connectivity to each ShadowNet node. We configure a logical router, which we call the *head* router of the ShadowNet node, to terminate these two connections. With the ShadowNet backbone connectivity provided by the ISP, all head routers form a full-mesh, serving as the core routers of ShadowNet. For Internet connectivity, the head router interacts with ISP’s border router, *e.g.*, announcing BGP routes.

Connecting device slivers on the same ShadowNet node can be handled by the switching layer of that node. The head routers are used when device slivers across nodes need to be connected. In ShadowNet, we make use of the carrier-supporting-carrier (CsC) capabilities provided by MPLS enabled networks. CsC utilizes the VPN service provided by the ISP, and stacks on top of it another layer of VPN services, running in parallel but isolated from each other. For example, layer-2 VPNs (so called pseudo-wire) and VPLS VPNs can be stacked on top of a layer-3 VPN service [15].

This approach has three key benefits. First, each layer-2 VPN or VPLS instance encapsulates the network traffic within the instance, thus provides strong isolation across links. Second, these are off-the-shelf production-grade services, which are much more efficient than manually configured tunnels. Third, it is more realistic from a configuration perspective, as we explain later. To emphasize, the layer-2 VPN and VPLS options that we heavily use in ShadowNet provides layer-2 connectivity similar to physical wires, meaning that any layer-3 protocol can run on top of it. For example, the data plane programmability provided by the routers that we deploy [18], allow for the realization of network layer protocols other than IP on these layer-2 services.

Figure 7 contains various examples of enabling connectivity, which we explain in detail next.

UsrLink: To instantiate a *UsrLink*, the instantiations of the two *UsrDevices* on the two ends of the *UsrLink* are first identified. We handle three cases, see Figure 7a). (We consider the *UsrLinks* connected to a *UsrSwitch* part of that *UsrSwitch*, which we describe later):

1) Two slivers are on the same physical device: for example, *VM1* and *VM2* are on the same server; *LR2* and *Head1* are on the same router. In this case, we use local bridging to realize the *UsrLink*.

2) Two slivers are on the same ShadowNet node, but not the same device: for example, *VM1* and *LR1*, *LR1* and *LR2*. We use a dedicated VLAN on that node for each *UsrLink* of this type. For example, *LR1* will be configured with two interfaces, joining two different

VLAN segments, one for the link to *VM1*, the other one to *LR2*.

3) Two slivers are on different nodes: for example, *LR2* and *LR3*. In this case, we first connect each sliver to its local head router, using the two methods above. Then the head router creates a layer-2 VPN to bridge the added interfaces, effectively creating a cross-node tunnel connecting the two slivers.

In each scenario above, the types of the physical interfaces that should be used to enable the link are decided, the selected physical interfaces are configured, and the resource usage information of the interfaces is updated.

The various MPLS VPN technologies achieve much higher levels of realism over software tunnels, because almost no configuration is required at the end-points that are being connected. For example, to enable the direct link between *LR2* and *LR3*, the layer-2 VPN configuration only happens on *Head1* and *Head2*. As a result, if the user logs into the logical router *LR2* after its creation, she would only see a “physical” interface setup in the configuration, even without IP configured, yet that interface leads to *LR3* according to the layer-2 topology. Furthermore, the rate-limiting filters are configured on the interfaces assigned to the head routers, leaving the user-exposed VM or router slivers with clean configurations.

User-view switches: Unlike for *UsrMachines* and *UsrRouters*, *ShadowNet* does not allocate user-controllable device slivers for the instantiation of *UsrSwitches*, but rather provide an Ethernet broadcasting medium. (See Figure 7b.)

To instantiate a *UsrSwitch* connecting to a set of *UsrDevices* instantiated on the same *ShadowNet* node, we allocate a dedicated VLAN-ID on that node and configure those device slivers to join the VLAN (*i.e.*, *LR5* and *LR6*). If the device slivers mapped to the *UsrDevices* distribute across different *ShadowNet* nodes, we first recursively bridge the slivers on the same node using VLANs, and then configure one VPLS-VPN instance on each head router (*i.e.*, *Head3* and *Head4*) to bridge all those VLANs. This puts all those device slivers (*i.e.*, *VM3*, *LR5*, *LR6*) onto the same broadcast domain. Similar to layer-2 VPN, this achieves a high degree of realism, for example on *LR5* and *LR6*, the instantiated logical router only shows one “physical” interface in its configuration.

Internet access: We assume that *ShadowNet* nodes can use a set of prefixes to communicate with any end-points on the Internet. The prefixes can either be announced through BGP sessions configured on the head routers to the ISP’s border routers, or statically configured on the border routers.

To instantiate a *UsrDevice*’s Internet connectivity, we

first connect the *UsrDevice*’s instantiation to the head router on the same node. Then we configure the head router so that the allocated prefix is correctly forwarded to the *UsrDevice* over the established link and the route for the prefix is announced via BGP to the ISP. For example, a user specifies two *UsrRouters* connecting to the Internet, allocating them with prefix $136.12.0.0/24$ and $136.12.1.0/24$. The head router should in turn announce an aggregated prefix $136.12.0.0/23$ to the ISP border router. Traffic filters are applied to the interfaces configured on the head routers to prevent packet spoofing and achieve rate-limiting.

4.2 Achieving isolation and fair sharing

As a shared infrastructure for many users, *ShadowNet* attempts to minimize the interference among the physical instantiation of different slices. Each virtual machine is allocated with its own memory address space, disk image, and network interfaces. However, some resources, like CPU, are shared among virtual machines, so that one virtual machine could potentially drain most of the CPU cycles. Fortunately, virtual machine technology is developing better control over CPU usage of individual virtual machines [5].

A logical router on a Juniper router has its own configuration file and maintains its own routing table and forwarding table. However, control plane resources, such as CPU and memory are shared among logical routers. We evaluate this impact in §6.3.

The isolation of packets among different *UsrLinks* is guaranteed by the physical device and routing protocol properties. For each *UsrLink*, we impose a default rate-limit (*e.g.*, 10Mbps), which can be upgraded by sending a request via the user-level API. We achieve rate limiting via hardware traffic policers [19] and Linux kernel support [4].

4.3 Enabling device access

Console or remote-desktop access: For each VM running on *VirtualBox*, a port is specified on the hosting server to enable Remote Desktop protocol for graphical access restricted to that VM. If the user prefers command line access, a serial port console in the VM images is enabled and mapped to a UNIX domain socket on the hosting machine’s file system [5]. On a physical router, each logical router can be configured to be accessible through SSH using a given username and password pair, while confining the access to be within the logical router only.

Though the device slivers of a slice can be connected to the Internet, the management interface of the actual physical devices in *ShadowNet* should not be. For example, the IP address of a physical server should be contained within *ShadowNet* rather than accessible globally. We thus enable users to access the device slivers through

one level of indirection via the ShadowNet controller.

Sniffing links: To provide packet traces from a particular `UsrLink` or `UsrSwitch`, we dynamically configure a SPAN port on the switching layer of a ShadowNet node so that a dedicated server or a pre-configured VM can sniff the VLAN segment that the `UsrLink` or `UsrSwitch` is using. The packet trace can be redirected through the controller to the user in a streaming fashion or saved as a file for future downloading. There are cases where no VLAN is used, *e.g.*, for two logical routers on the same physical router connected via logical tunnel interfaces. In this case, we deactivate the tunnel interfaces and re-instantiate the `UsrLink` using VLAN setup to support packet capture. This action, however, happens at the physical-level and thus is transparent to the user-level, as the slice specification remains intact.

4.4 Managing state

To extract the state of an instantiated `UsrMachine`, which essentially is a VM, we keep the hard drive image of the virtual machine. The configuration file of a logical router is considered as the persistent state of the corresponding `UsrRouter`. While a disk image can be attached to a newly instantiated VM, reviving the stored states, the router configuration files need additional processing. For example, a user-level interface may be instantiated as interface `fe-0/1/0.2` and thus appear in the configuration of the instantiated logical router. When the slice is deactivated and instantiated again, the `UsrInt` may be mapped to a different interface, say `ge-0/2/0.1`. To deal with this complication, we normalize the retrieved configuration and replace physical dependent information with user-level object handles, and save it as the state.

4.5 Mitigating and creating failures

Unexpected physical device failures can occur, and as an option we would like to be able to mitigate failures as quickly as possible to reduce user perceived down time. One benefit of separating the states from the physical instantiation is that we can replace a new physical instantiation with the saved state applied without affecting the user perception. Once a device or a physical component is determined to be offline, we identify all instantiated user-level devices associated to it. New instantiations are created on healthy physical devices and saved states are applied if possible. Note that certain users are specifically interested in observing service behavior during failure scenarios. We allow the users to specify whether they want physical failures to pass through, which is disabling our failure mitigation functionality. On the other hand, we also support failure injection, for example tearing down the physical instantiation of a link or a device in the specification.

For physical routers, the monitor performs periodic retrieval of the current configuration files, preserving the states of `UsrRouters` more proactively. When a whole physical router fails, the controller creates new logical routers with connectivity satisfying the topology on other healthy routers and apply the saved configuration, such as BGP setup. If an interface module fails, the other healthy interfaces on the same router are used instead. Note that the head router is managed in the same way as other logical routers, so that ShadowNet can also recover from router failures where head routers are down.

A physical machine failure is likely more catastrophic, because it is challenging to recover files from a failed machine and it is not feasible to duplicate large files like VM images to the controller. One potential solution is to deploy a distributed file system similar to the Google file system [13] among the physical machines within one ShadowNet node. We leave this type of functionality for future work.

5 Prototype Implementation

In this section, we briefly describe our prototype implementation of the ShadowNet infrastructure, including the hardware setup and management controller. (At the time of writing a four node ShadowNet instance is being deployed in a production network. We plan to include details of this deployment in the final paper.)

5.1 Hardware setup

We built two ShadowNet nodes and deployed them locally. Each node has two Juniper M7i routers running JUNOS version 9.0, one Cisco C2960 switch, as well as four HP DL520 servers. The M7i routers are equipped with one or two Gigabit Ethernet PICs (Physical Interface Cards), FastEthernet PIC, and tunneling capability. Each server has two gigabit Ethernet interfaces, and we install `VirtualBox` in the Linux Debian operating system to host virtual machines. The switch is capable of configuring VLANs and enabling SPAN ports. (The ShadowNet nodes being deployed in the production network are using very similar hardware.)

In the local deployment, two Cisco 7206 routers are used to emulate the ISP backbone. MPLS is enabled on the Cisco routers to provide layer-3 VPN service as the ShadowNet backbone. BGP sessions are established between the head router of each node and its adjacent Cisco router, enabling external traffic to flow into ShadowNet. We connect the network management interface `fxp0` of Juniper routers and one of the two Ethernet interfaces on machines to a dedicated and separate management switch. These interfaces are configured with private IP addresses, and used for physical device management only, mimicking the out-of-band access which is common in ISP network management.

5.2 Controller

The ShadowNet controller runs on a dedicated machine, sitting on the management switch. The controller is currently implemented in Perl. A Perl module, with all the user-level APIs, can be imported in Perl scripts to create, instantiate and access service specifications, similar to the code shown in Figure 5. A `mysql` database is running on the same machine as the controller, serving largely, though not entirely, as the persistent storage connecting to the controller. It saves the physical device information, user specifications, and normalized configuration files, etc. We use a different set of tables to maintain physical-level information, `phy_device_table`, and user-level information, `usr_link_table`. The Perl module retrieves information from the tables and updates the tables when fulfilling API calls.

The configuration effector of the ShadowNet controller is implemented within the Perl module as well. We make use of the NetConf XML API exposed by Juniper routers to configure and control them. Configlets in the form of parametrized XML files are stored on the controller. The controller retrieves the configuration of the physical router in XML format periodically and when `UsrRouters` are deactivated. We wrote a specialized XML parser to extract individual logical router configurations and normalize relative fields, such as interface related configurations. The normalized configurations are serialized in text format and stored in the `mysql` database associating to the specific `UsrRouter`.

Shell and Perl scripts, which wrap the VirtualBox management interface, are executed on the hosting servers to automatically create VMs, snapshot running VMs, stop or destroy VMs. The configuration effector logs into each hosting server and executes those scripts with the correct parameters. On the servers, we run low-priority `cron` jobs to maintain a fair amount of default VM images of different OS types. In this case, the request of creating a new VM can be fulfilled fairly quickly, amortizing the overhead across time. We use the following steps to direct the traffic of an interface used by a VM to a particular VLAN. First, we run `tunctl` on the hosting server to create a `tap` interface, which is configured in the VMM to be the “physical” interface of the VM. Second, we make use of `802.1Q` kernel module to create VLAN interfaces on the hosting server, like `eth1.4`, which participates in `VLAN4`. Finally we use `brctl` to bridge the created tap interface and VLAN interface.

Instead of effecting one configuration change per action, the changes to the physical devices are batched and executed once per device, thus reducing authentication and committing overheads. All devices are manipulated in parallel. We evaluate the effectiveness of these two

	Router	Machine	DB	Total
Default (ms)	81834	11955	452	94241
Optimized (ms)	6912	5758	452	7364

Table 2: Slice creation time comparison

heuristic in §6.1.

In our prototype, we simply configure all switch ports in trunk mode and allow all VLAN-ID on all ports. In this case, we do not need extra configuration if a particular VLAN is assigned to facilitate a `UsrLink` or `UsrSwitch`. We are in the process of implementing a more secure realization.

The device monitor module is running as a daemon on the controller machine. SNMP trap messages are enabled on the routers and sent over the management channel to the controller machine. Ping messages are sent periodically to all devices. The two sources of information are processed in the background by the monitoring daemon. When failures are detected, the monitoring module calls the physical-level APIs in the Perl module, which in response populates configlets and executes on the routers to handle failures. An error message is also automatically sent to the administrators.

6 Prototype Evaluation

In this section, we evaluate various aspects of ShadowNet based on two example slices instantiated on our prototype. The user specifications are illustrated on the left side of Figure 8; the physical realization of that specification is on the right. In *Slice1*, two locations are specified, namely LA and NY. On the LA side, one `UsrMachine` (M1) and one `UsrRouter` (R1) are specified. R1 is connected to M1 through a `UsrLink`. R1 is connected to the Internet through L2 and to R2 directly via L5. The setup is similar on NY side. We use minimum IP and OSPF configuration to enable the correct forwarding between M1 and M2. *Slice2* has essentially the same setup, except that the two `UsrRouters` do not have Internet access.

The right side of Figure 8 shows the instantiation of *Slice1* and *Slice2*. VM1 and LR1 are the instantiation of M1 and R1 respectively. `UsrLink L1` is instantiated as a dedicated channel formed by virtualized interfaces from physical interfaces, `eth1` and `ge-0/1/0`, configured to participate in the same VLAN. To create the `UsrLink L5`, ShadowNet first uses logical tunnel interfaces to connect LR1 and LR2 with their head routers, which in turn bridge the logical interfaces using layer-2 VPN. Note that in a different instantiation the head router may not be on the same physical router as LR1, in which case a VLAN would be used.

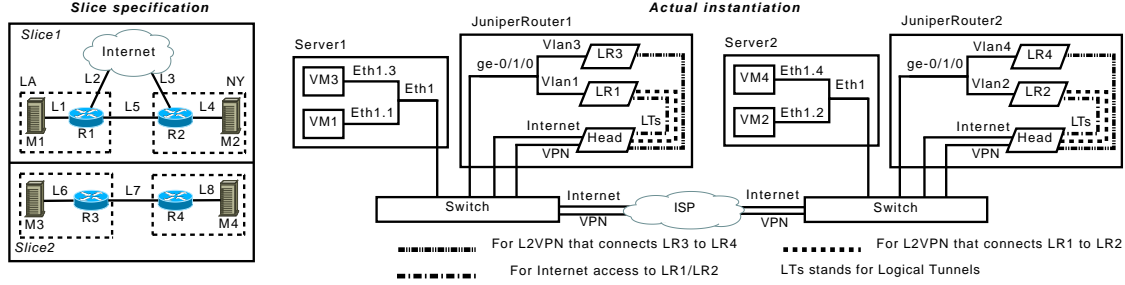


Figure 8: User slices for evaluation

bandwidth (Kbps)	packet size	Observed bandwidth	Delta (%)
56	64	55.9	.18
	1500	55.8	.36
384	64	383.8	.05
	1500	386.0	.52
1544	64	1537.2	.44
	1500	1534.8	.60
5000	1500	4992.2	.16
NoLimit	1500	94791.2	NA

Table 3: Cross-node link stress test

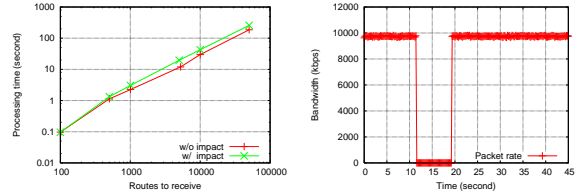
6.1 Slice creation time

Table 2 shows the creation time for *Slice1*, broken down into instantiation of machine and router, along with database access (DB in the table.) Using a naive approach, the ShadowNet controller needs to spend 82 seconds on the physical routers alone by making 13 changes, resulting a 94-second execution time in total. For machine configuration, two scripts are executed for creating the virtual machines, and two for configuring the link connectivity. With the optimization in §5, the total execution time is reduced to 7.4 seconds. Note that the router and machine configurations are parallelized, so that we have $total = DB + \max(Router_i, Machine_j)$. Parallelization ensures that the total time to create a slice does *not* increase linearly with the size of the slice. We estimate creation time for most slices to be within 10 seconds.

6.2 Link stress test

We perform various stress tests to examine ShadowNet’s capability and fidelity. We make L5 the bottleneck link, setting different link constraints using Juniper router’s traffic policer, and then test the observed bandwidth M1 and M2 can achieve on the link by sending packets as fast as possible. Packets are dropped from the head of the queue. The results are shown in Table 3, demonstrating that ShadowNet can closely mimic different link capacities.

When no constraint is placed on L5, the throughput achieved is around 94.8Mbps, shown as “NoLimit” in



(a) Impact of shared control (b) Hardware failure recovery planes

Figure 9: Control plane isolation and recovery test.

the table. This is close to maximum, because the routers we used as ISP cores are equipped with FastEthernet interfaces, which have 100Mbps capacity and the VM is specified with 100Mbps virtual interface. Physical gigabit switches are usually not the bottleneck, as we verified that two physical machines on the same physical machines connected via VLAN switch can achieve approximately 1Gbps bandwidth.

6.3 Slice isolation

We describe our results in evaluating the isolation assurance from the perspectives of both the control and data plane.

6.3.1 Control plane

To understand the impact of a stressed control plane on other logical routers, we run software routers, `bgpd` of `zebra`, on both M1 and M3. The two software routers are configured to peer with the BGP processes on LR1 and LR3. We load the software routers with BGP routing tables of different sizes, transferred to LR1 and LR3. The BGP event log on the physical router is analyzed by measuring the duration from the first BGP update message to the time when all received routes are processed.

In Figure 9(a), the bottom line shows the processing time of the BGP process on LR1 to process all the routes if LR3 is BGP-inactive. The top line shows the processing time for LR1 when LR3 is also actively processing the BGP message stream. Both processing times increase linearly with the number of routes received. The two lines are almost parallel, meaning that the delay is proportional to the original processing time. The difference of receiving 10k routes is about 13 seconds, 73 seconds for 50k routes. We have verified that the CPU

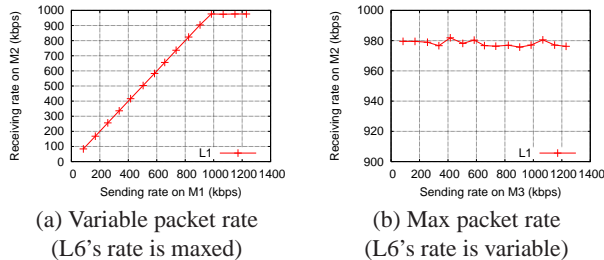


Figure 10: Data plane isolation test.

usage is 100% even if only LR1 is BGP-active. We have also used two physical machines to peer with LR1 and LR3 and confirmed that the bottleneck is due to the Juniper router control processor. We note that this property is only applicable to the control plane of Juniper M7i routers using “Internet Processor II”. We expect the impact is smaller when better control plane modules are used, for example the MX-series. If these limitations prove to be problematic in practice, solutions exist which allow a hardware separation of logical router control planes [17].

6.3.2 Data plane

L1 and L6 share the same physical interfaces, `eth1` on *Server1* and `ge-0/1/0` on *JuniperRouter1*. We restrict the bandwidth usage of both L1 and L6 to be 1Mbps by applying traffic policer on the ingress interfaces on LR1 and LR3. From the perspective of a given `UsrLink`, say *L1*, we evaluate two aspects: regardless of the amount of traffic sent on *L6*, (1) *L1* can always achieve the maximum bandwidth allocated (e.g., 1Mbps given a 100Mbps interface); (2) *L1* can always obtain its fair share of the link. To facilitate this test, we apply traffic policer on the ingress interfaces (`ge-0/1/0`) on LR1 and LR3, restricting the bandwidth of L1 and L6 to 1Mbps. Simultaneous traffic is sent from M1 via L1 to M2, and from M3 via L6 to M4.

Figure 10(a) shows the observed receiving rate on M2 (y-axis) as the sending rate of M1 (x-axis) increases, while M3 is sending as fast as possible. The receiving rate matches closely with the sending rate, before reaching the imposed 1Mbps limit. This demonstrates that *L1* capacity is not affected, even if *L6* is maxed out. Figure 10(b) shows the max rate of *L1* can achieve is always around 980kbps no matter how fast *M2* is sending.

6.4 Device failure mitigation

We evaluate the recovery time in response to a hardware failure in ShadowNet. While *Slice1* is running, M1 continuously sends packets to M2 via L1. We then physically yanked the Ethernet cable on the Ethernet module `ge-0/1/0`, triggering `SNMP LinkDown` trap message and the subsequent reconfiguration activity. Figure 9(b) shows the packet rate that M2 observes. The

downtime is about 7.7 seconds, mostly spent on effecting router configuration change. Failure detection is fast due to continuous SNMP messages, and similarly controller processing takes less than 100ms. This exemplifies the benefit of strong isolation in ShadowNet, as the physical instantiation is dynamically replaced using the previous IP and OSPF configuration, leaving the user perceived slice intact after a short interruption.

7 Related work

ShadowNet has much in common with other test/trial networks [10, 26, 21]. However, to our knowledge, ShadowNet is the first platform to exploit recent advances in the capabilities of networking equipment to provide a sharable, composable and programmable infrastructure using carrier-grade equipment running on a production ISP network. This enables a distinct emphasis shift from experimentation/prototyping (enabled by other test networks), to service trial/deployment (enabled by ShadowNet). The fact that ShadowNet utilizes production quality equipment frees us from having to deal with low-level virtualization/partitioning mechanisms, which typically form a significant part of other sharable environments. Please refer to §2.3 for comparison between ShadowNet and other networking experimental testbeds.

A similar service deployment incentive to that espoused by ShadowNet was advocated in [20]. Their service definition is, however, narrower than ShadowNet’s scope which also includes network layer services. Amazon’s EC2 provides a platform for rapid and flexible edge service deployment with a low cost [1]. This platform only rents computing machines with network access, lacking the ability to control the networking aspects of service testing, or indeed network infrastructure of any kind. PLayer [14] is designed to provide a flexible and composable switching layer in data-center environment. It achieves dynamic topology change with low cost; however, it is not based on commodity hardware.

Alimi *et al.* proposed the idea of shadow configuration [8], a new set of configuration files that first run in parallel with existing configuration and then either committed or discarded. The shadow configuration can be evaluated using real traffic load. The downside is that the separation between the production network and the shadowed configuration may not be strongly guaranteed. This technique requires significant software and hardware modification on proprietary network devices.

We heavily rely on hardware-based and software-based virtualization support [6] in the realization of ShadowNet, for example virtual machines [5] and Juniper’s logical router [16]. The isolation between the logical functionality and the physical resource can be deployed to achieve advanced techniques, like router

migration in VROOM [25] and virtual machine migration [12], which can be used by ShadowNet.

8 Conclusion

In this paper, we propose an architecture called ShadowNet, designed to accelerate network change in the form of new networks services and sophisticated network operation mechanisms. Its key property is that the infrastructure is connected to, but functionally separated from a production network, thus enabling more realistic service testing. The fact that production-grade devices are used in ShadowNet greatly improves the fidelity and realism achieved. In the design and implementation of ShadowNet, we created strong separation between the user-level representations from the physical-level instantiation, enabling dynamic composition of user-specified topologies, intelligent resource management and transparent failure mitigation. Though ShadowNet currently provides primitives mainly for service testing purposes, as a next step, we seek to broaden the applicability of ShadowNet, in particular, to merge the control framework into the production network for allowing service deployment.

References

- [1] Amazon Elastic Compute Cloud. <http://aws.amazon.com/ec2/>.
- [2] FEDERICA: Federated E-infrastructure Dedicated to European Researchers Innovating in Computing network Architectures. <http://www.fp7-federica.eu/>.
- [3] GENI: Global Environment for Network Innovations. <http://www.geni.net/>.
- [4] Traffic Control HOWTO. <http://linux-ip.net/articles/Traffic-Control-HOWTO/>.
- [5] VirtualBox. <http://www.virtualbox.org>.
- [6] K. Adams and O. Agesen. A comparison of software and hardware techniques for x86 virtualization. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006.
- [7] M. Agrawal, S. Bailey, A. Greenberg, J. Pastor, P. Sebos, S. Seshan, K. van der Merwe, and J. Yates. Routerfarm: Towards a dynamic, manageable network edge. SIGCOMM Workshop on Internet Network Management (INM), September 2006.
- [8] R. Alimi, Y. Wang, and Y. R. Yang. Shadow configuration as a network management primitive. In *Proceedings of ACM SIGCOMM*, Seattle, WA, August 2008.
- [9] H. Alzoubi, S. Lee, M. Rabinovich, O. Spatscheck, and J. V. der Merwe. Anycast CDNs Revisited. 17th International World Wide Web Conference, April 2008.
- [10] A. Bavier, N. Feamster, M. Huang, L. Peterson, and J. Rexford. In VINI veritas: realistic and controlled network experimentation. *SIGCOMM Comput. Commun. Rev.*, 36(4):3–14, 2006.
- [11] Cisco Systems. MPLS VPN Carrier Supporting Carrier. http://www.cisco.com/en/US/docs/ios/12_0st/12_0st14/feature/guide/csc.html.
- [12] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, 2005.
- [13] S. Ghemawat, H. Gobiuff, and S.-T. Leung. The google file system. *SIGOPS Oper. Syst. Rev.*, 37(5):29–43, 2003.
- [14] D. A. Joseph, A. Tavakoli, and I. Stoica. A policy-aware switching layer for data centers. *SIGCOMM Comput. Commun. Rev.*, 38(4), 2008.
- [15] Juniper Networks. Configuring Interprovider and Carrier-of-Carriers VPNs. <http://www.juniper.net/>.
- [16] Juniper Networks. Juniper Logical Routers. <http://www.juniper.net/techpubs/software/junos/junos85/feature-guide-85/id-11139212.html>.
- [17] Juniper Networks. Juniper Networks JCS 1200 Control System Chassis. <http://www.juniper.net/products/tseries/100218.pdf>.
- [18] Juniper Networks. Juniper Partner Solution Development Platform. <http://www.juniper.net/partners/osdp.html>.
- [19] Juniper Networks. JUNOS 9.2 Policy Framework Configuration Guide. <http://www.juniper.net/techpubs/software/junos/junos92/swconfig-policy/frameset.html>.
- [20] L. Peterson, T. Anderson, D. Culler, and T. Roscoe. A Blueprint for Introducing Disruptive Technology Into the Internet. In *Proc. of ACM HotNets*, 2002.
- [21] L. Peterson, A. Bavier, M. E. Fiuczynski, and S. Muir. Experiences building planetlab. In *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006.
- [22] L. Peterson, S. Shenker, and J. Turner. Overcoming the Internet Impasse through Virtualization. *Proc. of ACM HotNets*, 2004.
- [23] J. Turner and N. McKeown. Can overlay hosting services make ip ossification irrelevant? PRESTO: Workshop on Programmable Routers for the Extensible Services of TOMorrow, May 2007.
- [24] J. E. Van der Merwe et al. Dynamic Connectivity Management with an Intelligent Route Service Control Point. *Proceedings of ACM SIGCOMM INM*, October 2006.
- [25] Y. Wang, E. Keller, B. Biskeborn, J. van der Merwe, and J. Rexford. Virtual routers on the move: live router migration as a network-management primitive. *SIGCOMM Comput. Commun. Rev.*, 38(4), 2008.
- [26] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, 2002.