2014

A HyperNet Architecture

Shufeng Huang
University of Kentucky, theremetony@gmail.com

STUDENT AGREEMENT:

I represent that my thesis or dissertation and abstract are my original work. Proper attribution has been given to all outside sources. I understand that I am solely responsible for obtaining any needed copyright permissions. I have obtained needed written permission statement(s) from the owner(s) of each third-party copyrighted matter to be included in my work, allowing electronic distribution (if such use is not permitted by the fair use doctrine) which will be submitted to UKnowledge as Additional File.

I hereby grant to The University of Kentucky and its agents the irrevocable, non-exclusive, and royalty-free license to archive and make accessible my work in whole or in part in all forms of media, now or hereafter known. I agree that the document mentioned above may be made available immediately for worldwide access unless an embargo applies.

I retain all other ownership rights to the copyright of my work. I also retain the right to use in future works (such as articles or books) all or part of my work. I understand that I am free to register the copyright to my work.

REVIEW, APPROVAL AND ACCEPTANCE

The document mentioned above has been reviewed and accepted by the student's advisor, on behalf of the advisory committee, and by the Director of Graduate Studies (DGS), on behalf of the program; we verify that this is the final, approved version of the student's thesis including all changes required by the advisory committee. The undersigned agree to abide by the statements above.

Shufeng Huang, Student

Dr. James Griffioen, Major Professor

Dr. Miroslaw Truszczynski, Director of Graduate Studies

A HyperNet Architecture

DISSERTATION

A dissertation submitted in partial fulfillment of the
requirements for the degree of Doctor of Philosophy in the
College of Engineering
at the University of Kentucky

By
Shufeng Huang
Lexington, Kentucky
Director: Dr. James Griffioen and Dr. Kenneth L. Calvert, Professor
of Computer Science
Lexington, Kentucky
2014

ABSTRACT OF DISSERTATION

A HyperNet Architecture

Network virtualization is becoming a fundamental building block of future Internet architectures. By adding networking resources into the "cloud", it is possible for users to rent virtual routers from the underlying network infrastructure, connect them with virtual channels to form a virtual network, and tailor the virtual network (e.g., load application-specific networking protocols, libraries and software stacks on to the virtual routers) to carry out a specific task. In addition, network virtualization technology allows such special-purpose virtual networks to co-exist on the same set of network infrastructure without interfering with each other.

Although the underlying network resources needed to support virtualized networks are rapidly becoming available, constructing a virtual network from the ground up and using the network is a challenging and labor-intensive task, one best left to experts.

To tackle this problem, we introduce the concept of a *HyperNet*, a pre-built, pre-configured network package that a user can easily deploy or access a virtual network to carry out a specific task (e.g., multicast video conferencing). HyperNets package together the network topology configuration, software, and network services needed to create and deploy a custom virtual network. Users download HyperNets from HyperNet repositories and then "run" them on virtualized network infrastructure much like users download and run virtual appliances on a virtual machine. To support the HyperNet abstraction, we created a *Network Hypervisor* service that provides a set of APIs that can be called to create a virtual network with certain characteristics.

To evaluate the HyperNet architecture, we implemented several example HyperNets and ran them on our prototype implementation of the Network Hypervisor. Our experiments show that the Hypervisor API can be used to compose almost any special-purpose network – networks capable of carrying out functions that the current Internet does not provide. Moreover, the design of our HyperNet architecture is highly extensible, enabling developers to write high-level libraries (using the Network Hypervisor APIs) to achieve complicated tasks.

Keywords: HyperNet, virtual network, network hypervisor, programmable router, SDN

A HyperNet Architecture

By
Shufeng Huang

Dr. James Griffioen, Dr. Kenneth L. Calvert

Director of Dissertation

Dr. Miroslaw Truszczynski

Director of Graduate Studies

Jan. 28, 2014

ACKNOWLEDGMENTS

I am very thankful to all the people who had helped me to finish my doctoral study.

My thanks first go to my Ph.D. advisor Dr. James Griffioen. Thanks for advising and helping me becoming a better researcher in Computer Science. Thanks for arousing my curiosity about computer networking when I decided to pursue a Ph.D. Thanks for encouraging me whenever I encountered troubles in research. Thanks for spending time to review all my papers. Thanks for sending me to conferences where I got to know lots of people with the same research enthusiasm. I could not have achieved what I have achieved now without his professional guidance. I would also like to thank Dr. Kenneth Calvert for co-advising me. Every conversation during our weekly research project meeting was a great learning experience for me.

I would also like to thank the other members of my committee: Dr. Raphael Finkel, Dr. Robert Heath, and Dr. Hank Dietz for their expertise, criticism, availability and intellectually stimulating conversations. Special thanks to Dr. Finkel for guiding me with a research project when I first joined University of Kentucky.

I would also like to expressed my appreciation to Hussamuddin Nasir, Lowell Pike, William Marvel from the Lab for Advanced Networking for their excellent technical assistance with my projects, from machine maintenance, to help configure routers, to fixing technical bugs in testbeds, and other associated problems.

I had a pleasurable time in Kentucky with many fellow colleagues. Thanks for the stimulating discussions, knowledge and ideas, and collaboration on experimental research projects. Thanks for making life in the lab colorful.

I am also thankful to all my professors who had me in class and answered my questions. Thanks to all the staff in the CS department and in the Graduate school. You have all helped make my experience here at UK fulfilling.

Thanks to my family, my parents who raised me and sent me to the U.S., my uncle who introduced me to computer science when I was very young, and my grandma who always brings laughter to the family. I would not finish my doctoral study if it were not for your support.

Finally and most importantly, I would like to thank my beautiful, wonderful wife, Lei. You have carried the burden with me, and you have brought joy to my everyday life. I am lucky to have you, and it is only because of you that we made it through this part of our journey.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The huge success of the Internet over the past few decades has clearly demonstrated the wisdom of the early designers, who decided to create a simple best-effort packet delivery network and then couple that with (intelligent) programmable computers at edges of the network. One consequence is that applications running on end systems define the network's functionality not the network itself. This architecture enabled innovation on end systems that has allowed the Internet to be adapted and enhanced far beyond the imagination of the Internet designers.

However, it has become increasingly clear that certain innovations will require new functionality in the network itself. For example, due to the need for trustworthy communication, some argue that the Internet should provide intrinsic security in which the integrity and authenticity of communication is guaranteed [1]. Some argue that additional processing of packets should be provided within the network so that users can define their own processing, thereby choosing how their packets are processed by the network [2]. To support the increasing number of mobile end systems (mainly cell phones) on the Internet, some argue that the Internet should be re-designed to support network mobility at scale [3]. Yet there are other proposals addressing new network types such as vehicular networks [4] and ad-hoc networks [5]. Meanwhile, the existing Internet has several other recognized problems. Well-known Internet problems range from lack of address space [6] to routing problems [7, 8, 9],

to lack of support for mobility [10], to lack of security [11, 12, 13, 14, 15].

Although there are many arguments on how to realize a next generation Internet [16, 17], it is widely agreed that the future Internet should be highly flexible and programmable to support more innovation within the network itself. A promising technique that offers users the ability to program the network in safe, user-specific, ways is virtualization. Although virtual networks are beginning to emerge, the tools and interfaces for users to create virtual networks are severely lacking.

> *This thesis proposes a new HyperNet abstraction that simplifies the task of creating, deploying and using special purpose virtual networks – virtual networks that are tailored to the needs of particular applications, and are also tailored to the set of participants and users of the virtual network.*

1.1 Virtualization

Propelled by the need to efficiently and cost effectively support web services, *virtualization* technology has recently gained widespread popularity and use. Virtualization enables one to create multiple virtual instances of a device from a single physical device, allowing – for example – multiple virtual web server machines to be hosted on the same physical machine.

Virtual computers, referred to as *Virtual Machines (VMs)*, have not only become a key part of data centers hosting web services, but are now also commonly found on desktop users' machines. Virtual machines not only share the same set of physical resources (i.e., run on the same physical machine), but more importantly, VMs are also isolated from each other so that problems in one VM will not affect the execution of other VMs.

Virtualization has also changed the way people create, package, share and deploy software. The best example of this is a *virtual appliance*[18]. Virtual appliances

encapsulate *all* the pieces of a software system – including the operating system, application libraries, and configuration files – into a single package that can be easily run by users who would otherwise not be able to (or not want to) assemble and configure a complex software system. For example, bringing up and hosting a Content Management System (CMS) today no longer requires an expert to install, configure, and initialize the appropriate OS, web servers, databases, file systems, and CMS software. Instead, an average user can download a fully-configured and ready-to-run content management *appliance* from a virtual appliance "store" and simply "run" it on a virtual machine.

1.1.1 Virtual Networks

Recently the concept of virtualization has been extended from computers to network devices (e.g., network routers, switches, and links). Early examples of virtual routers were based on virtualized PCs acting as network routers. PlanetLab [19, 20], for example, allows users to reserve "slivers" (virtual machines) from physical machines scattered all across the world and connect them together via overlay links to form a "slice" (an overlay network). The virtual routers in the overlay can be programmed and controlled by users, enabling users to deploy custom code on the virtual routers in order to create application-specific networks. In Emulab [21], users can obtain real or virtual PCs from a cluster of resources for use as emulated network routers. The emulated routers are then connected using Virtual Local Area Networks(VLANs)[22]. Again, because users have complete control of the emulated routers, they can deploy their own protocols and network services to create a virtual network with customized functionality. The emerging GENI network [23, 24, 25] is perhaps the best example, offering a wide rang of (virtualized) network resources to create virtual networks that span the continent. Like PlanetLab and Emulab, users can design application-specific topologies and run application-specific code on the routers that comprise the network.

More recently, router vendors have begun to support virtualization of commercial network hardware. Examples include Juniper routers, which support the creation of multiple "logical routers" from a single physical router [26]. A variety of other router vendors now support OpenFlow [27] protocols and controllers. By leveraging existing tunneling technologies such as the Generic Routing Encapsulation (GRE) protocol [28] and the Multiprotocol Label Switching (MPLS) [29], users can create virtual links between virtual routers anywhere in the world. Moreover, one can easily imagine a future in which many, if not most, ISPs offer virtualized network resources for a fee.

The ability to reserve virtualized, and in some cases programmable, routers at locations all across national backbone networks and edge networks makes it possible for multiple wide area "virtual internets" to share the same underlying physical infrastructure and operate simultaneously without interfering with each other. Moreover, each of these virtual internets has the potential to use its own unique set of protocols and services tailored to a particular purpose. The power and flexibility of virtual networks makes it a perfect way to either try out new functionality or create and operate new services that are not possible today. In short, it is becoming increasingly clear that network virtualization will be a fundamental building block for future networks.

1.2 Building Virtual Networks

Although virtual networks bring many advantages, creating and building a virtual network is not an easy task.

1.2.1 Specifying the Network

As anyone who has used one of the existing virtual network infrastructures can attest, creating a fully functional virtual network is anything but easy. For example, in the

current GENI [23] environment, setting up a GENI experiment (a virtual network on GENI infrastructure) is an involved process consisting of several steps. The user must first discover the set of possible GENI network resources (e.g., routers, switches, PCs, and links.) that can be reserved. The user must then define the virtual topology by selecting a subset of the available resources to be used in the network, along with the network links that connect the resources together (specified in a Resource Specification ("RSpec") file). The user must then identify and load the software (e.g., operating systems, protocols, libraries, applications, services, etc.) that will provide the functionality for the virtual network. Once the software is loaded, the user must configure the system (as well as the software stacks) and launch necessary services. To run a particular application on the network, the user then needs to load and execute application-specific services and programs on the end system in the network.

While virtual machines can be configured with certain resources (e.g., memory or disk space), the number of configuration parameters is relatively small. A virtual network, on the other hand, can be instantiated in an endless number of ways depending on the resources (virtual routers and links) selected for inclusion. Determining which virtual routers to include in the virtual network and how those routers should be interconnected into a topology is a non-trivial process in and of itself. Moreover, having identified and allocated the virtual network resources, one must load, configure, and initialize the software for all the resources that make up the virtual network, which will differ from resource to resource. In short, creating a virtual network is a task best left to network experts.

1.2.2 Supporting New Functionality

While ISPs might see a niche market and decide to create a special-purpose network for a particular type of traffic, it is unlikely that the average user will create virtual networks tailored to their needs and users. Expertise is needed to create,

maintain, and operate virtual networks. Unfortunately, ISPs have little incentive to invest in a customized virtual network that does not have sufficient payback. For example, ISPs may invest in virtual networks to support widely-used applications of general interest to paying customers (e.g., a video conferencing network for large corporate customers), but they will have little desire to build networks for less profitable applications (e.g., a virtual network for a little-used application with few potential customers). Moreover, the application-specific network that ISPs deploy for high-margin applications will likely be long-running networks – permanent services available to all users. ISPs are unlikely to dynamically create a virtual network tailored to the specific users of a given session, but instead will support users in general (e.g., an ISP will create a single long-running video conferencing network used by all video conferences, as opposed to dynamically creating and deploying a network designed especially for the specific set of participants in some personal video conference). Application-specific virtual networks that are tailored only for small groups of participants and designed only for specific network applications will play a very important role in the future Internet and thus, new models are needed for an average user (instead of a network expert) to easily create and dynamically deploy a highly-tailored virtual network.

1.3 The HyperNet Approach

Ideally, to create a virtual network, one would like an abstraction similar in spirit to that of a virtual appliance, where all that one has to do to create their own virtual network is obtain a file containing a virtual network specification (i.e., the logical equivalent of a virtual appliance) and run it on virtual network infrastructure. Such a "virtual network" should contain the virtual network topology, the software and services needed by the network, and all the configuration information needed by the virtual network. All the pieces of a virtual network, along with the expertise needed

to combine them together, need to be encapsulated so that deploying such a virtual network is as easy as running a virtual appliance. The average user should not only be able to join a special-purpose virtual network, but the user should also be able to deploy his own specialized networks. While virtual machines have benefited from the concept and abstraction of a virtual appliance, the same cannot be said of virtualized network infrastructure where an analogous abstraction does not yet exist.

We take an approach similar in spirit to that of a virtual appliance, which we call a *HyperNet Package*.

In the simplest case, a HyperNet Package is used as follows: the user responsible for setting up the virtual network obtains a copy of the HyperNet Package[1] (say by downloading it from a "HyperNet App Store", much like users download virtual appliances). The user double-clicks on the downloaded file to "run it", which invokes a local program that executes the HyperNet Package. The user is then prompted for configuration information (assuming configuration information such as "a list of participants" is required and was not previously specified in a configure file). Executing the HyperNet Package results in communication with various virtual network infrastructure providers to allocate resources needed by the HyperNet Package, instantiation of the virtual network topology as specified by the HyperNet Package, and deployment of the appropriate software onto the reserved virtual routers. The HyperNet Package ultimately produces a functional virtual network waiting for users to join. Users that want to participate in the newly created virtual network must obtain the "end host" software – included in the HyperNet Package – and invoke it to join the virtual network. The end host must specify the virtual network identifier the end host wants to join (i.e., a unique "name" for the virtual network that is returned when the network was created) and any other credential information (e.g.,

[1]We use the term *HyperNet Package* to refer to the software package that can be downloaded and run, and *HyperNet Network* or *HyperNet Virtual Network* to refer to the resulting virtual network that is deployed when the HyperNet Package runs.

a "password") needed to join the network. As users join the HyperNet network, the HyperNet Package verifies that they are permitted to join, and their packets begin flowing across the virtual network. When the application session ends (which may be a few minutes or a few months or years later), the creator or other authorized individual tears down the virtual network.

In order to make virtualized networks usable to the largest audience possible[2], the HyperNet abstraction must be simple to use, yet powerful enough to support a wide range of applications. There are several challenging problems that must be addressed in the definition and implementation of such an abstraction including (1) determining where the abstraction fits into the overall network virtualization picture (which is still emerging), (2) defining the interfaces (APIs) used to launch a HyperNet Package and then access a HyperNet network, (3) deciding the set of capabilities a HyperNet Package should support, (4) addressing the topology generation issues associated with deploying a HyperNet network, (5) solving the participant joining problems (i.e., creating a convenient interface for Internet participants to join a virtual network), (6) ensuring the security of HyperNet networks, and (7) providing the building blocks needed to support a variety of economic eco-systems in which HyperNet networks can thrive. Throughout the remainder of this thesis, we will describe our solutions to these problems and present an initial prototype that demonstrates the feasibility of our approach.

1.4 Example HyperNet Packages

To illustrate the HyperNet abstraction, consider a HyperNet Package that enables any web server to deploy its own Content Distribution Network (CDN) tailored to the clients of that server. In our example, the CDN HyperNet Package decides where to place the content caches so that content is placed as close to the clients as possible.

[2]We envision three classes of users: *builders*, *creators*, and *participants*, which will be described later.

Unlike conventional CDNs that rely on DNS tricks at the source to redirect packets to CDN caches, the CDN HyperNet network operates at the IP level, running code on routers to intercept packets en route to the server and redirects them to a nearby CDN cache without DNS involvement. The HyperNet Package also includes an end system API that the web server can use to push content into the CDN network. The HyperNet Package might even come with a pre-installed and pre-configured web server that pushes content into the CDN by default.

One can imagine a wide variety of other example HyperNet Packages, each designed for a different purpose and set of users. To get a feel for the types of HyperNet Packages that might be useful, consider the examples listed below:

Audio/Video Net

An audio/video conferencing HyperNet Package would create a multicast infrastructure among the participants. The HyperNet Package might use a queuing strategy tailored to a specific codec that intelligently drops less important packets when network congestion arises, and thus improves the overall quality of service.

CDN Net

A CDN HyperNet Package would allow any user (typically a content provider) to create a special-purpose Content Distribution Network, storing the provider's content near the participants for fast lookup and retrieval.

Game Net

A game HyperNet Package would include a game server that provides minimum delay to all participants (game players) and create priority queues on each router, ensuring that packets carrying critical game actions receive higher priority than other packets.

Wireless Net

A wireless HyperNet Package, designed for wireless end systems, might contain code to be loaded into wireless access points to alleviate the problems that arise because of a weak signal or packet loss over the first/last hop wireless link.

Bit-escrow Net

A bit-escrow HyperNet Package would deploy code into routers to cache packets (i.e., "bit escrow") as they pass through the router. The cached packets would be stored only for a short period of time, but during that interval the router would respond to retransmission requests directly from the cache, thereby reducing the time required to recover from lost packets.

Backup Net

A backup HyperNet Package might create multiple paths between the client and the backup server to maximize the throughput across the network, thereby reducing the time required to complete a backup.

FS Net

A distributed (disconnected) file system HyperNet Package might deploy code along the paths between the participants to ensure file transfers are both reliable and efficient, transferring files hop-by-hop and supporting disconnected operation (e.g., a laptop that loses connectivity for some time).

Home Net

A home HyperNet Package would deploy code into the ISP's first hop router to prioritize traffic being sent over the limited-bandwidth channel coming into the home. This would allow home network users to ensure that low priority bulk transfers (e.g., Youtube video streams) do not interfere with higher priority traffic (e.g., Skype phone calls or interactive sessions like ssh).

None of these networks is particularly novel in and of itself. Similar types of special-purpose networks have been proposed in the context of active and programmable networks for many years[30, 31, 32]. What makes these networks interesting is not the fact that they can be constructed, but rather the way they are constructed. In particular, the goal of HyperNet Packages is to identify all code and expertise needed to create one of these networks and capture it in a self-contained object that can then be "unpacked" and deployed into an underlying virtual network infrastructure with little or no human (or application) intervention. As a result, even an average user can deploy special-purpose virtual networks.

1.5 Contributions of the Thesis

The contributions of our proposed HyperNet approach include:

- *A Virtual Network Package Abstraction:* Everything needed to deploy a virtual network is contained in a single **HyperNet Package**. The user simply needs to download the package and "run" it. As a result, even an average user can deploy virtual networks via the HyperNet abstraction.

- *Virtual Network Infrastructure Providers:* We define a *Virtual Network Infrastructure Provider (VNIP)* abstraction to capture the characteristics of new infrastructure providers that are beginning to emerge. We define a VNIP's services, its responsibilities, and also its relationship to other VNIPs and other components in the HyperNet architecture. We also define a standard interface for VNIPs to use to communicate with other components of the HyperNet architecture. We also give examples of emerging real-world virtual network infrastructure providers.

- *A Simple Interface:* The design of our HyperNet architecture offers an API that developers can use to easily compose an application-specific virtual network

package and distribute the package so that average users can deploy customized virtual networks.

- *Decoupling Service Providers from Resource Owners:* In our HyperNet architecture, the resource owners (e.g., today's ISPs) are no longer the only creators of the network. Instead, ordinary users can also become the *Network Creators*, creating virtual networks and providing network services. In fact, we expect more virtual networks will be created by ordinary users than by ISPs. Moreover, the virtual networks can be at small scale for short time durations, supporting a small group of participants. This will encourage smaller, lightweight, special-purpose virtual networks to appear, bringing new functionalities that have never been seen in the Internet.

- *A New Business Model:* We have already seen the huge success of Apple's App Store. We can think of the virtual network infrastructure provided by VNIPs as the HyperNet platform (akin to the IOS platform). With the help of a Network Hypervisor (akin to the IOS APIs), one can imagine a future where new HyperNet developers appear, creating additional HyperNet Packages to run on the HyperNet platform. Normal users can then buy HyperNet Packages from the HyperNet App Store and deploy them on VNIPs.

While the road to deployment and use is a long one, an ecosystem built around deployment and use of specialized virtual-network-based applications will encourage innovation from users, and show the way to a more adaptable next generation Internet that can change to meet the needs of future applications. As an initial step towards realizing a HyperNet architecture, this thesis describes the design of the HyperNet architecture in detail but leaves the security concerns and the economic infrastructure built on the HyperNet architecture as future work.

In the rest of this thesis, we begin by presenting related work in Chapter 2, including different kinds of virtualization technologies and past efforts made towards a "programmable Internet" such as active networks and service-centric networks. A description of Virtual Network Infrastructure Providers (VNIPs) is provided in Chapter 3, including a list of assumptions made of VNIPs and VNIP services. Chapter 4 gives an overview of the HyperNet architecture followed by a detailed description of the various components in the HyperNet architecture in Chapter 5, including the design of the Network Hypervisor service, the Network Hypervisor APIs and HyperNet Libraries, the process that a HyperNet Package takes to create and deploy a virtual network, and the participant usage models in the architecture. A prototype implementation of the Network Hypervisor service as well as some example HyperNet Packages are presented in Chapter 6 and Chapter 7, followed by some closing thoughts and conclusions in Chapter 8.

Chapter 2

Related Work

This chapter describes current and previous work related with HyperNets. Section 2.1 talks about virtualization technologies, including hypervisors, virtual machines, virtual appliances, commercial cloud services and platforms for building network services on the cloud. Section 2.2 introduces state-of-the-art techniques used in providing a virtual network infrastructure, including various tools used to facilitate the creation and management of virtual networks in GENI. The next two sections talk about previous approaches taken towards a "programmable network". Section 2.3 describes programmable network infrastructure, including past Active Networks research along with more recent projects such as NetServ, Service-centric Networks, and OpenFlow. Section 2.4 focuses on composable network stack approaches, including the X-kernel protocol stack, the Tau protocols, and the NC State Silo project.

2.1 Virtualization Technology

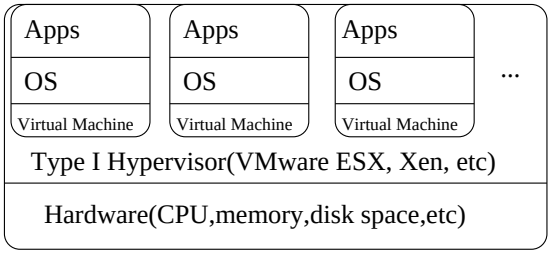
A variety of commercial and open source efforts now offer powerful and highly configurable virtualization solutions. Examples include VMWare, OpenStack, Citrix Xen, VirtualBox, and many others [33, 34, 35, 36, 37]. The idea of virtualization is to generate multiple instances of resources (computation power, memory, storage, etc) out of one physical device. These virtual instances of resources share the same

underlying physical device but yet are protected from one another. In the following section we briefly highlight some of the most common approaches to virtualization, pointing out their advantages and drawbacks.

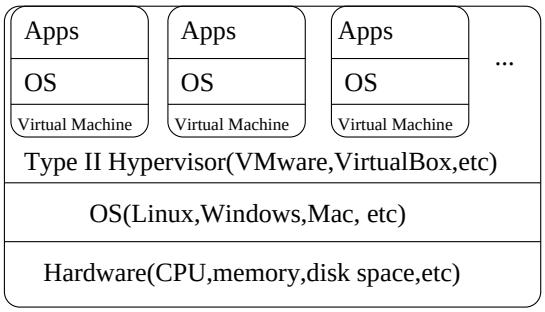
2.1.1 Hypervisors and Virtual Machines

Virtualization is typically achieved through a software layer called a **hypervisor** (or **virtual machine monitor**) running directly on the physical hardware (called a Type I hypervisor) or on a traditional operating system (called a Type II hypervisor). Hardware-assisted virtualization is also possible, but typically only appears in high-end servers. The hypervisor creates the illusion of multiple *virtual machines*.

A **Virtual Machine (VM)** [38] mimics the behavior of a physical machine, running its own operating system and applications much like a physical machine. Unlike a physical machine, a virtual machine does not have full access to all the resources on a physical device, but rather is sandboxed to a subset of the physical resources reserved for the VM.



(a) Type I Hypervisor



(b) Type II Hypervisor

Figure 2.1: Type I and type II Hypervisor model

Fig. 2.1(a) illustrates multiple virtual machines running on a Type I hypervisor while Fig. 2.1(b) illustrates multiple VMs running on a Type II hypervisor. Type I hypervisors run on the bare hardware, creating what looks like multiple instances of the bare hardware. Because OS and application code runs *unmodified* on the physical CPU, it requires that the hardware have the ability to catch privileged instructions and then reimplement them in the hypervisor. Type II hypervisors typically run on a "host OS". Before executing any application or OS code, the Type II hypervisor scans the binary code to find any privileged instructions and any instructions that cause branches/jumps. It then replaces those instructions with calls back to the hypervisor, the hypervisor then emulates the instructions in the context of the virtual machine that issued the instruction. Besides privileged instructions, a hypervisor also needs to emulate memory mapping (such as segmentation and paging) to support a VM.

- Virtual Machines (VMs), as we have defined them thus far, offer "true virtualization" in the sense that they try to mimic all aspects of a physical

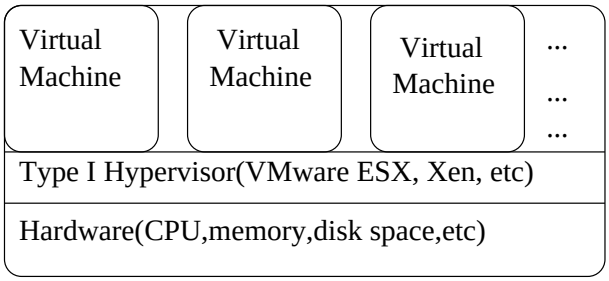


Figure 2.2: Virtual Machine

machine. As a result, they can be used to run any operating system (e.g., MS Windows, Apple OS X, Linux, etc). Examples of Type I hypervisors that offer this type of "true virtualization" include IBM's zOS [39] on its p and z-series machines, the Citrix XenServer [40] and the VMware ESX hypervisor architecture [41]. Examples of Type II hypervisors that offer "true virtualization" include VMware Workstation [42], VirtualBox [43] and KVM [44].

- A light weight form of virtualization that does not mimic a physical machine is *Operating System level Virtual Containers*. In this case the "containers" are

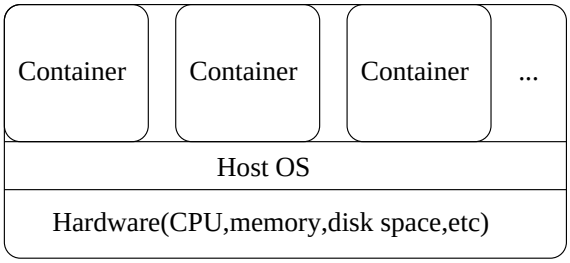


Figure 2.3: Virtual Containers

  isolated from one another, but provide an OS abstraction rather than a physical machine abstraction. As a result, containers only execute applications, not entire OSes. As shown in Fig 2.3, all containers sitting on top of the hosting operating system share the same OS kernel. A virtual container partitions processes and system state, replicates the file system, creates multiple file system roots, and replicates I/O devices to create pseudo devices. As a result, virtual containers create the look-and-feel of "virtualization" while underneath the containers is the same OS. Examples of virtual containers include BSD Jails [45], OpenVZ [34], and LXC [35].

Although virtualization allows users to run multiple systems on the same hardware, the user must install and set up the OS and all the software in every VM, which is a time consuming and often error prune task. To simplify the software installation setup and configuration steps, the virtualization community has developed the concept of a **Virtual Appliance (VA)**.

2.1.2 Virtual Appliances

A **Virtual Appliance** is a pre-built, pre-configured, ready-to-run application packaged along with the (optimized) operating system needed to run the application. The optimized operating system is sometimes called a JeOS, or Just enough operating

system, because it contains only the OS features needed to support the specific application. In other words, JeOS is a super light weight OS tailored only for the use of a specific application. Thus, compared with a general-purpose operating system, a JeOS requires a much smaller footprint, fewer patches, and is more secure and easy to maintain.

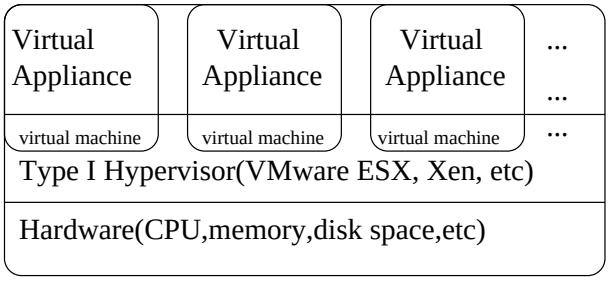


Figure 2.4: Virtual Appliances built on Type I Hypervisor

Fig. 2.4 shows multiple Virtual Appliances running on the same hardware. By comparing this figure with Fig. 2.1, we can see that a Virtual Appliance is basically a pre-built, pre-configured super-slim JeOS plus an application (stack) running on the JeOS[1].

2.1.3 From Virtual Machines to Virtual Infrastructure

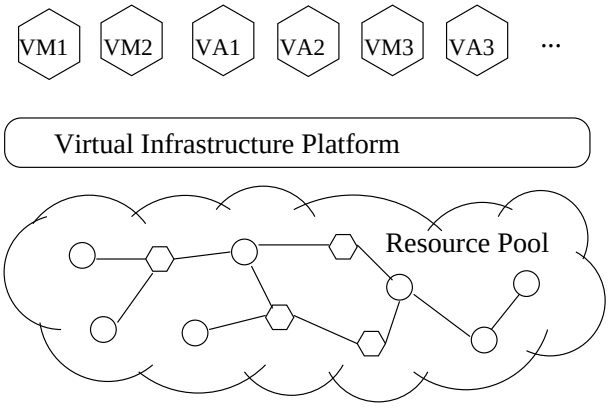


Figure 2.5: Virtual Infrastructure

[1]As an example, an Ubuntu-based Drupal Virtual Appliance [46] installation package is no more than 170M Bytes in size.

To increase availability and resilience of a virtualized system, one can use multiple physical devices to implement the virtualization. **Virtual Infrastructure** is a software layer built on a pool (set) of physical resources (including servers, storage, and network[2]) that enables the dynamic sharing of resources among multiple appliances, as depicted in Fig. 2.5. A virtual infrastructure aggregates physical servers along with the networks and storage into a unified pool of physical resources that can be utilized by appliances when and where they are needed. It has the following components:

- A Type I hypervisor on each server (machine)
- Virtual Infrastructure services such as resource management and consolidated backup to optimize available resources
- Automated solutions that provide special capabilities to optimize a particular IT process such as provisioning or disaster recovery.

Examples of virtual infrastructure services include:

- *vMove*: moving running virtual appliances from one physical server to another without user knowledge.
- *Distributed Resource Scheduler*: monitoring utilization across resource pools and intelligently aligning resources with business needs.
- *High availability*: in cases when one server is down, appliances will fail over to another server without being noticed by users.
- *Consolidated backup*: achieving efficient backup by offloading the backup workload from production VMs to proxy servers in the virtual infrastructure.

[2]Note that in a virtual infrastructure, the network is only used to enable the migration of appliances. There is no (or limited) explicit sharing of network resources (i.e., routers and switches) in virtual infrastructure at present.

2.1.4 Other Virtualization Approaches

Interestingly enough, if we say virtual machines realize virtualization in a "bottom-up" fashion (the same set of hardware supports the execution of multiple software stacks, including operating systems), there are other virtualization approaches that are taken in a "top-down" fashion. The most well-known example is a Java Virtual Machine (JVM). A JVM is a virtualized software platform used to execute Java programs. Every Java program is first compiled into an intermediate language called Java bytecodes that is interpreted and executed by the JVM. Interpreting bytecodes is slow. To improve the performance of Java, most Java systems support a Just In-Time Compiler (JIT) that translates Java bytecode into native machine language that can be executed directly on the physical machine. The .NET Framework compiles and executes C++/C#/VB code in the same way as JVM. First, code is converted to the Microsoft Intermediate Language (MS-IL) — akin to Java bytecodes. Then by using a JIT compiler similar to JVM, the Common Language Runtime (CLR) of the .NET Framework converts MS-IL to native machine language. These virtualization approaches enable one to execute the same code on hardware regardless of the operating system or the hardware being used. In some way, this leads to the idea of executing code on routers, which later on becomes one of the efforts made towards a programmable network – the Active Networks Approach, which will be described in Section 2.3.

2.1.5 Cloud Services

The term *cloud* is used as a metaphor to depict the Internet as an abstraction of the virtualized resources it contains. "The Cloud" refers to a shared pool of computing resources, storage services and higher level applications/services that can be accessed over the Internet (say, via a web browser) to perform a service for users on-demand. Despite various concerns about security, users have been rapidly embracing "the

cloud" because of the convenience it offers, such as on-demand software, resource pooling, broad network access, rapid elasticity, and measured service. Traditionally, a user would purchase a license from a software provider and then install and run the software on a dedicated on-site server (or servers). With cloud computing, users can now purchase software services from the cloud on demand and use those services via the Internet. This allows users to avoid dealing directly with licensing costs, hardware costs, and maintenance costs, while allowing access to the service from anywhere. Cloud computing service providers maintain their pool of computing resources in such a way that it is very convenient for users to expand their reservations (purchase more resources) to accomplish their tasks, thus providing rapid elasticity. Usage is measured, which is analogous to how traditional utility services are consumed[3]. Studies have already proven that cloud computing can help reduce IT budgets [47].

The services provided by cloud computing can be categorized into three groups: *software as a service (SaaS)*, *platform as a service (PaaS)*, and *infrastructure as a service (IaaS)*. SaaS provides a way for users to run their applications over the Internet from centralized servers rather than from on-site servers. Traditionally, users purchase licences from software providers and then install and run the software in a dedicated server. With SaaS, users pay the software provider a subscription fee for the service. The software is hosted from the software provider's servers and is accessed by the users over the Internet. Google Docs [48] is an example of SaaS. PaaS refers to products that are used to deploy applications. PaaS offers users a way to access applications provided by platform providers themselves or third party providers. Google AppEngine [49] is an example of PaaS. IaaS is the essence of cloud computing. It is the base for both SaaS and PaaS. Infrastructure providers provide the physical storage space and processing capabilities in the cloud. This category

[3]There is some confusion about cloud computing, grid computing, and utility computing. Generally speaking, grid computing focuses on how a group of computers cooperate to finish a huge task; utility computing focuses on packaging computer resources as a metered service.

includes cloud storage, managed hosting, and development environment that allow users to build applications. Amazon's EC2 [50] is an example of IaaS.

In short, cloud computing provides users a cheaper and easier way to manage their applications than using the traditional model. Due to the large amount of resources in the cloud, cloud computing can be potentially optimized to boost the execution of applications and thus is more efficient than the traditional model. However, the overall performance of cloud computing depends not only on the efficiency of the computing part, but also relies on the efficiency of the networking part. In other words, if the Internet does not provide fast, reliable, Quality-of-Service (QoS) guaranteed delivery, the overall performance of cloud computing will be reduced. Moreover, it is the users' responsibility to configure the resources in the cloud to build any new services, which is a hard task that is typically left to experts. Big companies will only build new services with mass appeal.

2.1.6 Deploying Cloud Services

Cloud computing makes it easier and cheaper for service providers to provision computing resources and to flexibly extend resources on the fly. Virtual appliances make it easy to configure software stacks on a single node. However, it is still a challenging and time-consuming task when it comes to building and configuring an entire network infrastructure which contains many nodes. Chef [51] is designed to facilitate the creation and deployment of cloud services. In Chef, rules for configuring each node to reach a pre-defined state are expressed in the form of a "run list". A node might be an application load balancer, an application server, an application database cache, an application database, or a monitoring node. Each node of the infrastructure is pre-loaded with a Chef-client. The Chef-client is in charge of fetching the run list of that node from a centralized Chef-server. In addition, the Chef-client also ensures that the run list of a node is achieved in compliance with the policy set

by the Chef-server. A run list is a list of "recipes" and a recipe simply states which application (or "resource") a node should install, what configuration a node should have, etc. Recipes are stored in "cookbooks" to realize code re-use and modularity. In addition, Chef also provides a "search" function so that the configuration of the service infrastructure can adapt automatically with the addition or removal of nodes to/from the infrastructure. As a result, to setup and configure a cloud service, the user only needs to define policies and recipes in the Chef-server.

In some sense, Chef is trying to achieve the same goal as HyperNets: to ease the process of building and deploying a virtualized infrastructure. However, the HyperNet architecture is different from Chef in the following ways. First, Chef only deals with cloud service infrastructure and thus, is more specialized. In other words, Chef only cares about the infrastructure used within a network service (a network service infrastructure might include multiple service load balancers, applications, databases and database caches). How end systems reach the service infrastructure is not Chef's concern. The HyperNet architecture, however, can be used to create a virtual network that covers the end-to-end communication between any two end systems (participants) in the network. Secondly, the HyperNet architecture tries to answer the question of "how to tailor a network to match the requirement of a specific type of application and a specific set of participants" and thus is designed with APIs to help a HyperNet builder achieve this. Chef, on the other hand, only provides interfaces for one to customize the inner structure of a cloud service.

2.2 Virtual Networks

More recently, the concept of virtualization has been extended from computers to networking devices. Emerging network testbeds allow users to reserve virtualized network devices that act like virtual routers, and connect them together with virtual links to form a virtual network. The GENI [23] network is probably the best example

of a virtual network provider that is available to users today. This section describes some of the virtual network infrastructures available in GENI and the current tools provided by each of them.

GENI (Global Environment for Network Innovation) is a testbed network designed to give researchers the opportunity to experiment with new network protocols and architectures at scale. Currently GENI includes the following control frameworks: ProtoGENI [52], PlanetLab [20], ExoGENI (previously known as ORCA) [53] and ORBIT [54]. Because GENI is still under development, the list of tools that users can leverage to build their experimental network ("slice" in GENI terminology) is rather limited. However, there are tools and services that assist with resource discovery and allocation, tools to load software onto the nodes ("slivers") in a slice, and tools to help instrument and monitor an experiment. In the following, we introduce some of the GENI control frameworks and the corresponding tools that help users managing resources in those control frameworks.

2.2.1 ProtoGENI

ProtoGENI is one of the control frameworks that supports the GENI Aggregate Manager (AM) API [24]. Its predecessor is Emulab [21]. Each virtual network instance in ProtoGENI is called a "slice" and each programmable node in a slice is called a "sliver". A sliver may be a virtual machine or a physical machine. ProtoGENI uses an XML file called a "RSpec (Resource Specification)" file to request resources, describe the topology, and configure (programmable) routers. By using ProtoGENI API calls, users upload their RSpec files to ProtoGENI, which in turn allocates, sets up, and deploys the virtual networks specified in the RSpec file. Currently ProtoGENI provides API calls to: (1) discover available resources; (2) query information about resources; (3) create/delete/update/renew a sliver; (4) register/unregister/shutdown/renew a slice; and (5) check the status of a sliver/slice.

Network resources in ProtoGENI are grouped into aggregates spread across locations in the United States.

Flack [55] is a web-based GUI originally designed to create slices in ProtoGENI [52]. Flack has since been extended to interoperate with other types of aggregates including PlanetLab, ORCA and Openflow [56]. Flack's interoperability is due in large part to its support for the GENI Aggregate Manager (AM) API [24] which allows interoperability across different types of GENI aggregates. Flack provides a listing of all the resources available at the various aggregates. Using Flack, users can "drag" resources onto a canvas, linking them together to form the topology for their experimental network (slice). The output of Flack is an RSpec describing the topology. The RSpec can then be given to the aggregate managers who will reserve, allocate, and then initialize the requested resources. Flack provides users with control over the type of operating system used on nodes and the types of bandwidth and delay offered by the links that make up the topology. Flack also allows users to provide a tar file containing software to be loaded onto the node when it is initialized.

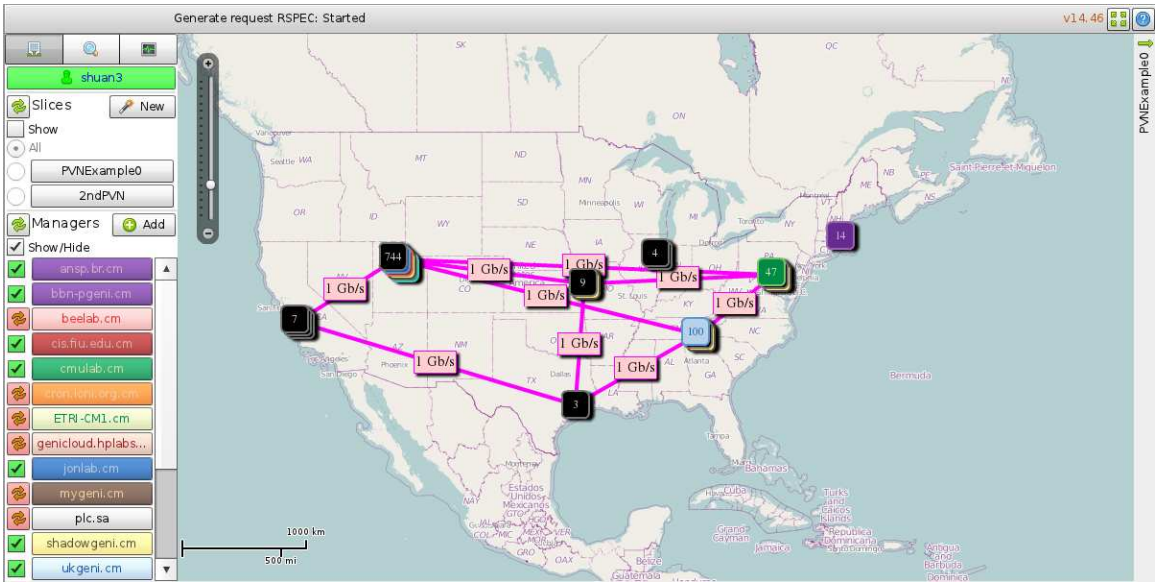


Figure 2.6: Flack Interface

The most significant advantage of the Flack interface is that it gives the user a

visualized idea about where (geographically) the resources are reserved, what types of resources are reserved (for both virtual routers and virtual links), and what the logical topology looks like, as shown in Fig 2.6.

Although the Flack interface makes it easy for users to create virtual networks and even load software on the nodes in a virtual network, it does not help users determine which resources should be included in the network, nor does it help setup or configure the network. Users still need to manually log onto each reserved node and do node-specific configuration (e.g., change routing tables, configure software, run node-specific commands, etc.) after the virtual network is created via Flack. Moreover, Flack only helps to allocate and connect GENI resources together, users must manually create channels/tunnels between non-GENI resources and the GENI resources allocated to a slice. Since the majority of users are located on non-GENI nodes/resources, there is a need to support a regular Internet user who wishes to connect to and use GENI.

2.2.2 ORCA

ORCA [53] is another GENI Control Framework. It uses Flukes [57], a Java-based GUI similar to Flack, to create, inspect, and manage experiments on ORCA. Flukes features are similar to Flack, except that unlike Flack, Flukes uses ORCA's native resource description language (NDL-OWL) to manage its resources as opposed to GENI RSpecs. Recent enhancements to ORCA have enabled it to connect RSpecs to NDL-OWL, but at its base level, ORCA still uses NDL-OWL. By using the NDL-OWL interface, users are able to specify things not available in RSpecs, e.g., to define and configure node groupings, to specify functional dependencies between nodes by configuring node boot sequences, or to specify post-boot script templates. However, like Flack, experimenters are still responsible for much of the configuration and setup after the resources have been allocated and become available for use.

2.2.3 PlanetLab

PlanetLab consists of a collection of machines hosted by members of the PlanetLab consortium. All PlanetLab machines run a common software package that includes a Linux-based operating system. It has mechanisms for bootstrapping nodes and distributing software updates. Tools like the *PlanetLab User Shell (plush)* [58], a command line tool, can be used to set up and deploy experiments in PlanetLab. The *PlanetLab Experiment Manager* tool [59] is used to simplify the deployment, execution and monitoring of experiments. It also supports an XML-RPC interface that allows other experimenter tools and services to be developed. The *PlanetLab Application Manager* [60] is used to help deploy, monitor and run applications on Planetlab. The application manager manages, installs, upgrades, starts/stops, monitors and controls an application. *Gush* [61] and *Raven* [62] are examples of tools being developed to support experimentation in PlanetLab. The goal of Gush is to provide an extensible execution management system for GENI. The implementation of Gush was recently extended to also support ProtoGENI.

Two main components in the Gush architecture are the controller and the clients. The Gush controller runs on a user's end system and the Gush clients run on each of the GENI nodes accepting commands from the controller. The controller receives and responds to inputs provided by the user via a CLI and configuration scripts. Users describe their experiment in an application specification XML document, and, based on this document, Gush locates, contacts, and prepares resources for use. Gush users can easily load specific software onto nodes as well as execute commands either from the Gush CLI or from configuration scripts. Because each Gush client runs a heart-beat protocol with the Gush controller to monitor the health of each GENI node, the Gush system is able to quickly report a failed node and find a replacement for that node.

Unlike the previous tools, Gush does not help to create the RSpec (i.e. to define the

slice or its topology), but rather assumes this has been done with some other tool. In order for a user to leverage Gush's software loading capabilities, the user must write a Gush script and create the associated tar files using Gush's XML-based configuration language. Finally Gush, like the other tools, lacks support for connecting to and interacting with real world users on non-GENI nodes.

Raven [62] (previously Stork [63] in Planetlab) is another tool in GENI that is specifically designed to support long-term experiments. It provides configuration management tools and resource management tools for long-term experiments where both software and resources can change during the lifetime of the experiment. It provides interfaces for administrators to enter instructions that will be applied to each individual system. Just like Gush, Raven also provides helper tools to assist experimenters in monitoring and managing their experiments.

Although some additional tools have begun to emerge for GENI, there is a significant learning curve required to utilize them effectively. Moreover, the decisions about topology and resource allocation are still very visible to the experimenter. While the topology is important, the details of the topology often are not important to the experimenter. For example, if a user wants to develop a new interactive multiplayer gaming network with a centralize controller, the user will not care about the specific nodes chosen to be part of the network. Instead, the user will simply want to know that the topology connects the various players of the game (participants) to a node that is centrally located relative to the participants. In other words, the user would like to be able to select the "type of topology", not necessarily know the details.

With the existing GENI tools, the experimenter is responsible for identifying, selecting, and then loading, all the software needed by the network. Networks are complex systems with multiple layers of software. In most cases, experimenters have no desire to (re)create all these software layers. Instead they would like to leverage existing software and services to the greatest extent possible, only modifying or

enhancing the layer of software that is the focus of their experiment. In other words, they would like to select an existing software stack, and have that stack automatically deploy as the software system for the experimental network.

Finally to get real-world participants to use an experimenter's new network requires support to "connect them into the slice" even though they are on machines that are not GENI-enabled. In other words, experimenters would like the ability to include non-GENI-enabled nodes in the topology. This typically requires some sort of tunneling over IP to link these nodes into the GENI network.

2.3 Programmable Network Infrastructure

Beginning in the mid 90's, a significant amount of research was directed toward the goal of creating a network infrastructure that is programmable. Much of the research was focused on programming abstractions, especially languages, for expressing application-specific processing to be carried out in the network.

2.3.1 Active Networks

The existing Internet architecture pushes all functionality to end nodes and keeps the network as simple as possible. The idea behind active networks, however, is to let custom programs be executed within the network, on routers. The idea first emerged in the 90's when the computational power of network devices dramatically increased and people realized that they can put more computational tasks into routers and switches. Generally speaking, there are two major advantages of using active networks: (1) they enable a wide range of new applications that leverage computation in the network and (2) they accelerate the pace of network innovation by separating service from the underlying infrastructure.

A *capsule-based* active network pushes the programmability of active networks to an extreme: every active packet (or capsule) in the network carries programming

code (or instructions) that can be read and executed by intermediate forwarding nodes which in turn change the functionality of the network. The ANTS toolkit [64], designed by Wetherall et al., allows end users to send out capsules containing any program the user wants. A capsule includes an ANTS-specific header immediately following the IP header (since capsules do not want to stop non-activated nodes from forwarding them). Besides version, type and the actual type-specific programming code fields, the ANTS header also includes the previous address to enable an active node to go back to the previous active hop to fetch program code. The ANTS approach assumes there are no programs in the active nodes initially. A lightweight code distribution system pre-transfers the programs to active nodes. The code-distribution control plane is separated from the data plane of capsules. The type field in the ANTS header is actually an MD5 hash of the corresponding program code. Thus, a type identifier names an implementation, not an interface. The previous address field enables program code delivery even when capsules (and their code) are created on-the-fly. The combination of the above two ensures that capsules can be carried by reference and also loaded on demand.

Many Active Network papers talk about how active nodes should be designed and programmed. Some focus on how to make the router flexible in terms of extending their services for active networks. Some focus on how to build an Execution Environment (EE) that provides more security, fine-grained IO control, and is easier for composing service. Others focus on innovative active services and active applications that can be built in an Active network, with quite a few arguments on whether putting active applications on top of active services is better or letting both be executed directly on a well-designed EE is better. The research produced a wide range of active network platforms, execution environments, active services and active applications. To deploy an active application/service, we need to first decide which execution environment should be used, which might in turn rely on

some particular active network platforms. This layered stack makes the maintenance and configuration of active networks complicated.

Active networks have not achieved widespread adoption for a number of reasons. First, the security concerns associated with executing arbitrary user code in the network have discouraged ISPs from enabling programmability in their networks. Second, running a program every time a packet arrives at a router introduces significant performance issues. Third, it is unclear how to charge users for this feature. Emerging virtual network infrastructure avoids or answers some of the problems that have plagued active networks. Because slices are private and isolated, security concerns are greatly diminished. Because slice resources are reserved/purchased, billing can be more easily integrated.

2.3.2 NetServ

The NetServ [65] Project is a collaboration between Columbia University, Bell Labs, Deutsche Telekom Laboratories and DoCoMo Labs Europe. The purpose of this project is to provide an extensible architecture for core network services for the next generation Internet. The key idea of NetServ is service modularization. By using NetServ, the functions and resources on a network node are divided into small and reusable building blocks. New network services can be composed and implemented by combining the functionality of building blocks available from multiple network nodes. To efficiently manage the building blocks (or service modules which are composed of multiple building blocks), a virtual services framework is created as one of the key pieces of the NetServ Architecture. The framework offers security, portability across hardware platforms, and the ability to control resource allocation among modules. Moreover, the framework supports adding and removing service modules at runtime [65]. The NetServ group implemented a prototype version of their network using a Click modular router [66] as the base router platform and

a Java-based dynamic module system OSGi [67] as the virtual services framework. They showed that using Java as the execution environment can result in tolerable processing time delays.

2.3.3 Service-Centric Networks

Service-centric networks [68, 69], proposed by Wolf et al., base the communication abstractions of the future Internet on the processing (including transfer) of information rather than the process of sending data. By giving the network some clues about the semantics of the information that is to be transferred, the network can provide more advanced services as opposed to blindly forwarding bits. In their Information Transfer and Data Services (ITDS) architecture, (by having processing as a first-class networking feature) an end system application can specify the *information transfer* that is desired and the network can then determine the appropriate handling of the data. A list of initial information transfer characteristics is proposed in [68]: Streaming vs. Random Access, Point-to-Point vs. Multi-Point, Interactive vs. "Canned", Bandwidth-Sensitive vs. Delay-Sensitive. These characteristics can together determine a packet flow class. Based on these characteristics, a network node with processing power can decide to receive, store, process (modify), and transmit the data. Before an end-to-end connection is set up, end systems communicate with a service controller. The service controller maps communication requirements onto multiple service nodes. When the required service is unavailable, the service controller passes the request to neighbor service controllers. To make the whole service more scalable, a hierarchy of service controllers may be created. However, since the service requires that state be maintained for each flow on each service node, there are potential scalability issues that are not addressed by the authors.

2.3.4 OpenFlow

OpenFlow [27] explores the power of providing a programmable control plane. It allows a network manager to specify how flows of packets (defined by an n-tuple of fields from a packet such as the ingress interface, source IP, destination IP, IP protocol, source port, destination port, and IP type of service) are handled (e.g., dropped or forwarded to a specific port) at each OpenFlow switch. The OpenFlow protocol defines a control mechanism through which the flow classification and forwarding tables in an OpenFlow switch can be dynamically configured by an *OpenFlow controller* (a centralized management component in an OpenFlow network). Due to its scalability constraints, OpenFlow is typically used in campus networks or enterprise networks. Although OpenFlow eases the task of managing the network from controlling every packet of the network in a distributed way (the Active Network approach) to controlling network packets in units of flows on a centralized controller, the user still needs to write OpenFlow controllers that deal with each specific flow.

2.4 Composable Network Stacks

Active networks try to control/program the network at the granularity of one packet. This fine-grain control gives the user a lot of flexibility but at the same time, adds a lot of complexity to the system. Composable Network Stacks try to support composable/tailored protocol stacks. Their goal is complete flexibility in the design of the protocol. Several of these systems attempted to automate the process of stack composition – automatically adjusting the stack to the needs of the current application or service.

2.4.1 X-Kernel Protocol Stacks

X-Kernel [70] focuses on layering and efficient implementation of communication protocols. Three primitive communication objects are defined in the X-Kernel:

protocols, sessions, and messages. Protocol objects can be static or passive; common examples include the protocols currently used in the IP stack. Session objects are passive but can be generated dynamically, representing connections. Message objects are active objects that move through the session and protocol objects in the kernel. A set of support routines facilitate the implementation of protocols. For example, the *buffer manager routines* manipulates messages. The map manager routines maintain bindings from identifiers (such as those extracted from message headers) to kernel objects. The event manager routines manage the timing of any procedure (e.g., so that a protocol can timeout). Last, the X-Kernel provides infrastructure to support communication objects. The relationship between communication objects can be represented by a simple textual graph description language. The "relationship graph" can be read by a composition tool which in turn generates the C code that creates and initializes the protocols described in the graph. Each object is implemented in a object-oriented style, i.e., each object has pointers to object-specific functions. More specifically, the protocol object can create a session by itself (the *open* function), or can pass its capability to a lower level protocol and ask it to create a session (the *open_enable* function), or tell its upper level protocol that it has created a session on its behalf (*open_done* function). In addition, the protocol object can also pass a message to one of its sessions via the *demux* function. The session object can either pass a message down to a low-level session via the push function, or pass a message up to a high-level session via the pop function (called by the *demux* function). A user-space process is also treated as a protocol in the X-Kernel protocol stack design, i.e., the user must export those operations that a protocol or session may invoke.

2.4.2 Tau protocols

Layering has been widely acknowledged as an efficient way to make protocols more efficient. However, it has also been identified as a performance impediment

because it requires that messages be processed sequentially. The Tau (Transport and up) protocol [71, 72], proposed by Calvert et al., achieves modularity in protocol implementations. They focus on end systems, and thus, transport layer and higher protocols. The term protocol module is used to refer to the object that implements the functions defined by a protocol specification. The idea includes two parts, explicit metaheaders and generic interfaces. Explicit metaheaders is an extension of the metaprotocol in O'Malley and Peterson's architecture in the sense that in the simplest case, it adds one byte in the header (of course, the metaheader could be more complicated than one byte), indicating the total number of headers in the packet. Since each header has fixed size, this metaheader contains sufficient information to permit the protocol headers on a data unit to be located all at once. To isolate protocol functions from the details of the infrastructure, a small number of generic interfaces is created. Five classes of interfaces are defined. The Data Outgoing (DO) interface is used whenever user data goes from one module to another on its way down to the transmitting substrate. The Data Incoming (DI) interface is used for user data that comes in the opposite direction. The Header Outgoing and Header Incoming interfaces are the analogues of DO and DI for headers. The Control-Only (CTL) interface is used to pass local control and synchronization information (such as passing a pointer to per-session state information to another module). This separation of control functions from data-manipulation protocol functions enables each protocol function module (except for data-handling functions) in Tau to operate independently and concurrently on any given message. Modules are no longer stacked unless they needed to be. Thus, Tau achieves high execution performance, especially when the system has multiple cores.

2.4.3 The SILO Project

The SILO [73, 74] Project focuses on cross-layer (or cross-service) interactions. The design goals of SILO are (1) construct a framework of fine-grain building blocks along with explicit support for combining elemental functions in a highly configurable manner to achieve flexibility and extensibility; (2) use a fine-grained modularization of networking functions to enable a scalable, unified Internet; (3) explicitly build in the ability for functional blocks to interact with each other to facilitate cross-service interactions; (4) treat security functions as easily pluggable components to smoothly integrate security features; (5) offload small but computationally intensive functions to secondary CPUs to take advantage of new performance-enhancing techniques.

"Services" are the fundamental building blocks in SILO. A service is a well-defined atomic function performed on application data that accomplishes a specific communication task. This fine-grained definition provides more flexibility compared with current protocols which typically embed complex functionality. Any services can be selected to accomplish a particular task, but the order that those services are applied is not tied to layers; instead, it is tied to a set of more general precedence constrains. SILO separates a service and its implementation, which the authors call a method (so that multiple methods could be associated with one service). A method that implements a service must implement the service-specific interfaces (as described in the service specification), as well as any service-specific knobs. A silo is actually an ordered subset of methods, each of which represent a different service. In the SILO architecture, a control agent in each node is responsible for composing such a silo. Besides composing a silo, a control agent is also in charge of adjusting all service and method-specific knobs to facilitate cross-service interactions. SILO also defines a minimum set of precedence constrains: Requires, MustOccurAbove, MustOccurImmediatelyAbove, MustNotOccurImmediatelyAbove to guide the service composition. By explicitly requiring each method to provide a

minimum control interface, and a minimum set of precedence constraints [74], SILO bypasses the potential unmanageable and unmaintainable consequences caused by cross-layer interaction.

2.5 Summary

The use of virtualization has made the development, deployment and management of applications a lot easier than the traditional model. Developers no longer need to worry about platforms and hence can focus on features of the applications they are developing. Virtual Infrastructure has enabled the sharing of a network of resources and the migration of virtual appliances among them. Currently the focus of virtual infrastructure is on enterprise server networks and the resources it deals with are mostly virtualized servers and storage (network is assumed to be robust and is used as a reliable connectivity provider among servers). Although the design of a HyperNet Package is not intended to facilitate a virtual infrastructure, the concept of a Network Hypervisor service can also be used in building a delivery network for virtual infrastructure.

Programmable Network Infrastructure and Composable Network Stack proposals show previous efforts towards making a programmable Internet. They are not directly related to the design of the HyperNet architecture but are great examples of potential platforms for building/operating HyperNet Packages. Moreover, they also suggest what a future programmable network device might look like or be capable of. We are not interested in dynamic composition of protocols as used in SILO because it is extremely challenging to compose an optimal stack on the fly. Instead, we want experts to compose a set of stacks and include them in a HyperNet Package along with some simple rules describing when each of the stacks should be used. Moreover, the HyperNet Package should not only include stacks, but also applications. Thus, the HyperNet Package is a complete package which contains both the protocol stacks

necessary to build a network (or network services) and the applications that will use them.

Chapter 3

Virtual Network Infrastructure Providers (VNIPs)

Before describing our HyperNet Architecture, it is important to take a moment to understand the evolving network virtualization efforts and the roles ISPs will play in offering virtualization services in the future. In order for HyperNet networks to provide end-to-end support between participants, we must understand the challenges and possibilities of the emerging virtual network providers.

We envision a future where there exist providers who offer virtual network infrastructure for a fee (or possibly some other form of compensation — say the right to monitor your traffic and sell that information to advertisers). We call these providers *Virtual Network Infrastructure Providers (VNIPs)*. Because network virtualization is still a relatively young technology, there is not much standardization across the various virtualization systems. While we fully expect that some type of standardization will occur over time – the GENI Aggregate Manager API [24] being a prime example of such an effort – we do not expect that all network virtualization infrastructure will look the same, offer the same set of services, have the same API, or operate under the same set of policies. Thus, we expect that each VNIP will offer a different set of services and have a different API for accessing those services. However, in order to create end-to-end networks, it must be possible for a HyperNet Package to reserve and use resources from heterogeneous VNIPs. Consequently, we need to

make some basic assumptions of all VNIPs. We will discuss our required common set of services in the later part of this chapter.

We envision two types of VNIPs. The first type of VNIP is a *Hardware Infrastructure Provider (HIP)* who owns and operates physical network hardware (routers, switches, PCs acting as routers, wired and wireless channels, etc) that can be virtualized and assigned to different virtual networks.

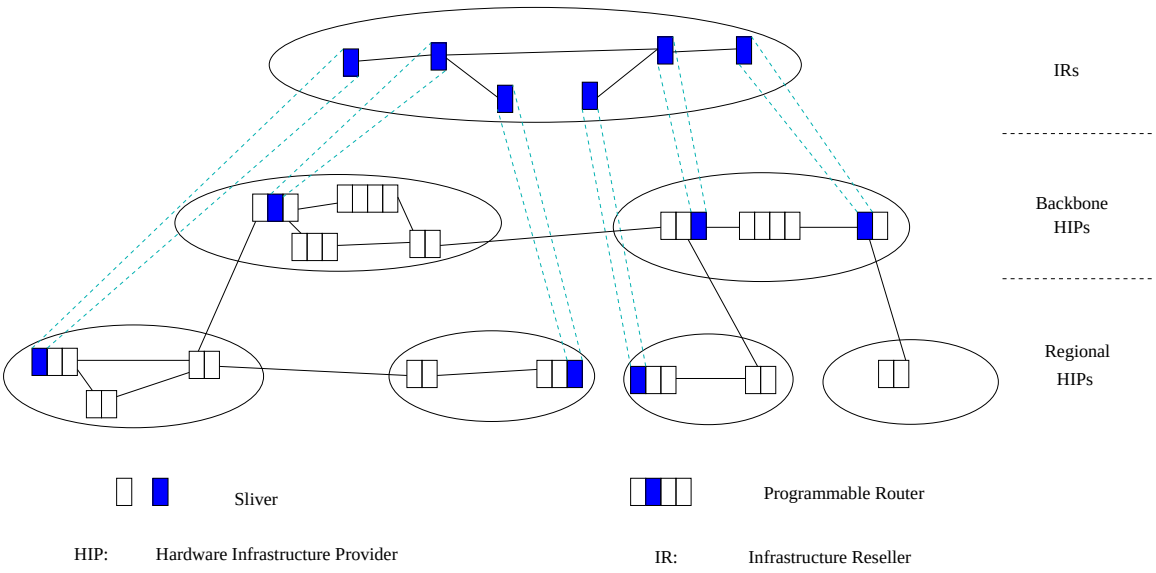


Figure 3.1: Types of virtual network infrastructure providers: HIPs and IRs

The second type of VNIP is an *Infrastructure Reseller (IR)* who does not own the hardware, but rather purchases the virtual network resources from hardware infrastructure providers (HIPs) and then resells them. Just as a travel agent does not own airplanes but resells flights from various airlines to create a complete flight for a customer, IRs can compose and resell HIP resources in more appealing/useful ways. Fig. 3.1 illustrates the two types of VNIPs: HIPs and IRs. A "sliver" in the graph is simple a virtual node (along with the resources) reserved out from a physical node.

To abstract the types of devices for use by a HyperNet network, we define four types of virtualized resources that a hardware infrastructure provider can make available for use:

- *Programmable Routers (PRs)* can be programmed via one of several standardized interfaces[1]. PRs may be virtual or physical routers, but are reserved for use by the HyperNet network. Here "Programmable" means the authorized end users (or the HyperNet Package that acts on their behalf) can load any program on the PR and run it. This gives the HyperNet Package complete control over the processing of packets that pass through the PR. Many of the current VNIPs (e.g., GENI, PlanetLab, Emulab) provide such programmable routers (typically in the form of a Linux PC).

- *Way Points (WPs)* are non-programmable routers that a user can only tunnel packets through en route to a PR or end system. An example of a way point could be an OpenFlow switch whose forwarding table can be tweaked by the HyperNet Package to forward (tunnel) network packets to some next hop. Just like a programmable router (PR), a Way Point may be virtual or physical, but they are exclusively reserved for use by the virtual network created by the HyperNet Package. Unlike a PR, a way point only allows the HyperNet Package to configure the routing table (or forwarding table) in the way point. A way point is not "programmable" by the end user. If we consider programmable routers as a way of controlling how each packet in the network is processed, then the way points can be seen as a light-weight way of controlling how network packets (typically in units of flows) are routed. Compared with programmable routers, Way Points provide a light weight way of defining paths – which is often all that is needed.

- *Programmable Servers (PSs)* are virtualized resources offered by the VNIP that provide computation and storage at locations inside the network. Programmable servers typically have huge amounts of disk space and high

[1]For now we assume that a programmable router provides a Unix-like user interface, and if granted, users have superuser access to install or run any program on it. Other interfaces may also be supported in our future HyperNet implementations.

processing power, and are capable of dealing with service requests and sending back results. The platform provided by a Programmable Server can vary, from an OS-level abstraction to an application-as-a-service abstraction. For example, the platform might provide a content management service interface, or a web hosting service interface, or possibly a bare virtual machine on which users can install a customized operating system.

- *Links.* Links may be physical cables, fibers or wireless channels that "physically" connect resources together, or they may be virtual channels. VNIPs allow links to be allocated and set up for the HyperNet Package to connect resources owned by the VNIP. VNIPs must also support links that connect to other VNIPs. Note that some VNIPs will own hardware that is not directly connected. For example, PlanetLab consists of hardware nodes spread across the world and interconnects its infrastructure using "links" that traverse the existing TCP/IP Internet (i.e., IP tunnels). In this case the VNIP is using conventional IP routers as resources in the topology, but we assume these are completely hidden/unobservable by the user of the VNIP (i.e., the HyperNet Package). Examples of such links would be GRE (Generic Routing Encapsulation) tunnels. While it is unlikely that there exists a physical channel from every virtualized router to every other virtualized router, it is possible (in fact likely) that a VNIP (HIP or IR) would offer full $N \times N$ connectivity between virtual routers via virtual channels, thereby significantly increasing the number of potential paths and indirectly complicating the HyperNet Package's task of selecting a optimal topology for the virtual network. Also, note that, although physical links based on IP have no ability to offer QoS, virtual channels between virtual routers may be able to offer QoS guarantees, which need to be reserved during the network creation phase.

Much like the current Internet, we envision several tiers of hardware infrastructure

providers, some offering long-haul backbone virtualized network resources, others offering regional resources, and still others offering local virtualized resources. Creating a virtual network will generally involve obtaining and linking together resources from several of these infrastructure providers.

3.1 Assumptions about VNIPs

In order for HyperNet Packages to be able to communicate with and reserve resources from VNIPs, we need to make some basic assumptions about VNIPs and the APIs they provide. These assumptions are not critical and could be relaxed or modified if VNIPs were to change or converge to a standard in the future. Given that current VNIPs are connected to the existing TCP/IP Internet, we assume that the Internet Protocol (IP) is supported by all VNIPs and can be used to identify and access the (virtual) resources offered by VNIPs, including physical nodes, end systems, PRs and WPs. In other words, in our architecture, IP is guaranteed to be available for users, VNIPs and hypervisor servers to communicate with each other. However, assuming IP does not imply the software that will be loaded onto the virtual nodes has to use IP protocols. For the HyperNet Package, we only use IP as the control channel to identify and communicate with the resources (i.e., to deliver HyperNet Packages and software Images, and to control HyperNet virtual networks). The actual data channel can use HyperNet-specific protocols.

To help HyperNet Packages discover appropriate resource to include in the virtual network, we assume that a VNIP knows:

- *Location Information about the resources.* This location information can be represented by an IP address or a geographical location or even a covered area (e.g., a zip code or a city/building name).

- *Static information about its resources*, such as the resource type (virtual/physical router, virtual/physical PC, etc) and resource capability (total amount of CPU/memory/disk, link capacity, etc).

- *Dynamic information about its resources*, such as current available bandwidth, current delay/loss rate of a physical link. That information can be monitored by substrate resources and kept in a local repository [75]. The VNIP may further summarize the information into long term average statistics and store them in a centralized database.

  It is not critical that this information be real time, but the more accurate it is, the better HyperNet Packages will be at selecting good resources to use.

- *Physical topology information about its resources.* Knowing the physical topology can help a HyperNet Package allocate the most appropriate resources and can avoid issues of placing multiple virtual resources on the same physical resource. The physical topology can be queried by the HyperNet Package but the VNIP may apply its own policy regarding how much information to give back to customers.

3.2 Basic VNIP Services

We assume that there are a set of basic functionalities that all VNIPs provide, including:

- API calls to reserve and free resources.

- API calls to connect the reserved resources together to form a virtual network. In our design, a VNIP provides two API calls to reserve virtual links:

  $$Tunnel\ createTunnel(Node\ a,\ Node\ b);$$

  $$Tunnel\ createTunnel(Node\ a,\ Node\ b,\ NodeSet\ tunnel\_points);$$

The *createTunnel()* call returns a tunnel connecting Node a and Node b. If no tunnel points are provided, it is up to the underlying VNIP to decide how to connect the two nodes. Alternatively, user can specify the intermediate hops of a tunnel by defining the tunnel points between Node a and b. There can be two types of tunnels returned: a *base tunnel* and a *composite tunnel*.

  - *Base Tunnel*: A base tunnel is a tunnel that appears to be a direct connection, with no information about intermediate hops. There are three types of base tunnels: (1) a physical link direct connecting two neighboring nodes. (2) an overlay tunnel (e.g., GRE tunnel) that travels through the Internet, connecting two programmable nodes. (3) a tunnel that travels through multiple hops of the underlying VNIP. However, the VNIP (for privacy reasons) does not want the user to know the two nodes are not directly connected to one another.
  - *Composite Tunnel*: A composite tunnel has either user-specified or user-discoverable (in this case, the tunnel is created without specifying the tunnel points) tunnel points within the tunnel. A user can issue further calls to discover the tunnel points within the composite tunnel.

- API calls to discover the physical topology. This facility enables HyperNet packages to learn what resources are available and how they are (physically) connected.
- API calls to start, stop, and renew (the reservation) of a virtual network.
- API calls to pay for usage of the resources.
- API calls to detect errors/failures in the infrastructure.

VNIPs may apply different information hiding policies. For example, one VNIP may choose to expose its full physical topology to its users while another VNIP may only give back a partial topology upon a "discover topology" request from its user.

3.3 VNIP API

To provide the basic VNIP services described above, each VNIP may provide its own API calls for its users to use. Although the appearance (i.e., name, parameters and return types) of the API calls may differ from one another, in our design, we assume that a set of common VNIP API calls can be summarized for every VNIP to follow. In fact, the GENI community is making a big effort towards such a common set of API calls [76]. The GENI AM API works well across several different GENI control frameworks including Planetlab, ProtoGENI, InstaGENI and ExoGENI. Our architecture simply leverages the GENI AM API as the VNIP API.

Chapter 4

HyperNet Packages

Emerging virtualization infrastructure, be it virtualized cloud resources or virtualized network resources such as routers, switches, and links, requires a new abstraction to efficiently program and utilize the infrastructure. In the following, we introduce a new abstraction, called a *HyperNet Package*, that enables average users to effectively use these new and emerging programmable infrastructures.

4.1 The HyperNet Abstraction

The *HyperNet* abstraction is inspired, in many ways, by *virtual appliances* [77]. A virtual appliance (section 2.1.2) encapsulates the components and configuration of a purpose-built software system into an "appliance" that can be run on a virtual machine (VM). The virtual appliance configuration specifies the necessary characteristics of the VM needed to execute the virtual appliance, while the packaged software components include everything from the operating system and networking stacks to libraries to applications, as well as all their configurations. The result is a single package that can easily be "run" by normal users who would otherwise not be able (or want) to assemble such a complex system. For example, anyone who has tried to set up a content management system (CMS) from scratch knows that it is a tedious process and takes a great deal of expertise and knowledge. Because of virtual appliances, bringing up and hosting a CMS today no longer requires the expertise

to install, configure, an initialize the appropriate OS, web servers, databases, file systems, and CMS software. Instead, a normal user can download a fully configured and ready-to-run content management "appliance" from a virtual appliance "store" and simply run it in a virtual machine. All the expertise is contained in the virtual appliance.

Because virtual appliances run on VMs, they are highly portable and can be run on any platform that supports VMs. While this advantage is indeed a consequence of virtualization, it is not necessarily the main contribution of virtual appliances. Instead, one can argue that the key contribution of the virtual appliance abstraction is its ability to bundle software and expertise into a package that can be run by the average user who would otherwise not be able to set up such a complex system. Unfortunately, a similar abstraction for virtualized networks does not exist. Ideally, one would like a similar abstraction, where all that is needed to run a custom, application-specific network is to obtain the appropriate virtual network "appliance" (i.e., file/package) and "run" it.

Given such an abstraction, one could imagine companies or individuals putting together a wide range of special-purpose virtual networks customized for a particular application or application domain, or designed to support a particular set of service requirements (QoS), network sizes, sets of participants, etc.

To illustrate the HyperNet abstraction, consider a video conferencing network, in which a conference organizer downloads a HyperNet Package from a web site (e.g., a "HyperNet market", similar to Apple's "app store" concept), and uses it to create a virtual network specifically designed for video conferencing and tailored to the participants of her conference. For example, the HyperNet Package might be designed for MPEG video [78] and include code (deployed on virtual routers at strategic locations) that gives priority to packets containing I frames over those containing

P and B frames[1]. It might be equipped with congestion marking techniques that provide explicit feedback to endpoints so that the video transmission can be adjusted by the end system applications [79]. Moreover, because it is designed for a specific set of participants, the HyperNet Package can build source-specific or shared multicast tree topologies, and then automatically deploy code in routers to implement multicast routing and forwarding protocols. The HyperNet Package may also include code to track the virtual network usage for each individual participant for billing purposes. The key point is that the average user can easily "deploy" a personal, highly tailored virtual network and begin using it right away. It no longer requires an expert to deploy rather complicated special-purpose networks.

One can easily imagine a wide variety of HyperNet Packages, each designed for a different purpose. Examples include: *Video Net* that multicasts live video streams to its subscribers; *Game Net* that guarantees small delay between the game server and each player; *Content Distribution Net* that picks content caches near the participants; *Home Net* that prioritizes the incoming and outgoing traffic traveling though the limited-bandwidth home gateway. Note also that a HyperNet Package needs not provide any new or custom services. A HyperNet Package may simply implement a regular IP network, but one that is private to (i.e., exclusive for) its users.

None of the HyperNet Package examples shown above are novel in or of themselves. Similar ideas have been proposed in the context of active and other programmable networks for some time. The key is not the ability to create such special-purpose virtual networks, but rather *the way those networks are created*. Our goal is to make it possible to package the expertise of network architects and designers into self-contained HyperNet Packages so that non-expert users can easily create virtual networks as if they were experts. HyperNet Package developers define the structure

[1]I frames are intra frames, which contain the information of a full picture in a video and can be decoded independent of P and B frames. Thus, an I frame is more important in MPEG video decoding than P and B frames which only include information about changes from other frames.

of the virtual network and the software to run on the virtual routers. As a result, the average user can simply download and use a HyperNet Package without having to know how the network is setup, configured, or operated.

4.2 Establishing Context

The first thing a virtual appliance must do is to establish the context it needs in order to offer its services. For example, a virtual appliance that acts as firewall, or an intrusion detection system, or a network nanny begins by specifying the characteristics of the virtual machine (VM) on which it must be run. In particular, the appliance will need at least two network interfaces: one facing the Internet and one facing the local area network (LAN). A virtual appliance that offers a video game experience needs to ensure that the VM has the necessary graphics/display capabilities needed to offer the game.

A similar problem arises in the context of HyperNet Packages. When a HyperNet Package is about to start up, it needs to create a topology from the available resources which involves discovering the available resources, selecting the ones to be used, and initializing them. It also means providing abstractions/methods by which the participating parties can access/use the newly created virtual network. In other words, a HyperNet Package must contain code that can create a virtual network topology from the available resources. Once created, the HyperNet Package needs to load application-specific software onto the resources, configure the software and start it. The HyperNet Package must load different types of software on the nodes depending on whether they are routers or end systems. Finally the HyperNet Package must be able to "watch" or "monitor" the resulting network to react to any failures or changes that need to occur (e.g., a new participant wants to join the network). In short, a HyperNet Package is essentially a program that establishes the virtual network and monitors it. The question is "what does this program look like?" and

"where does it run?". In the following section we describe the HyperNet architecture including the design of a HyperNet Package and the platform (Network Hypervisor) on which it is run.

4.3 The HyperNet Architecture

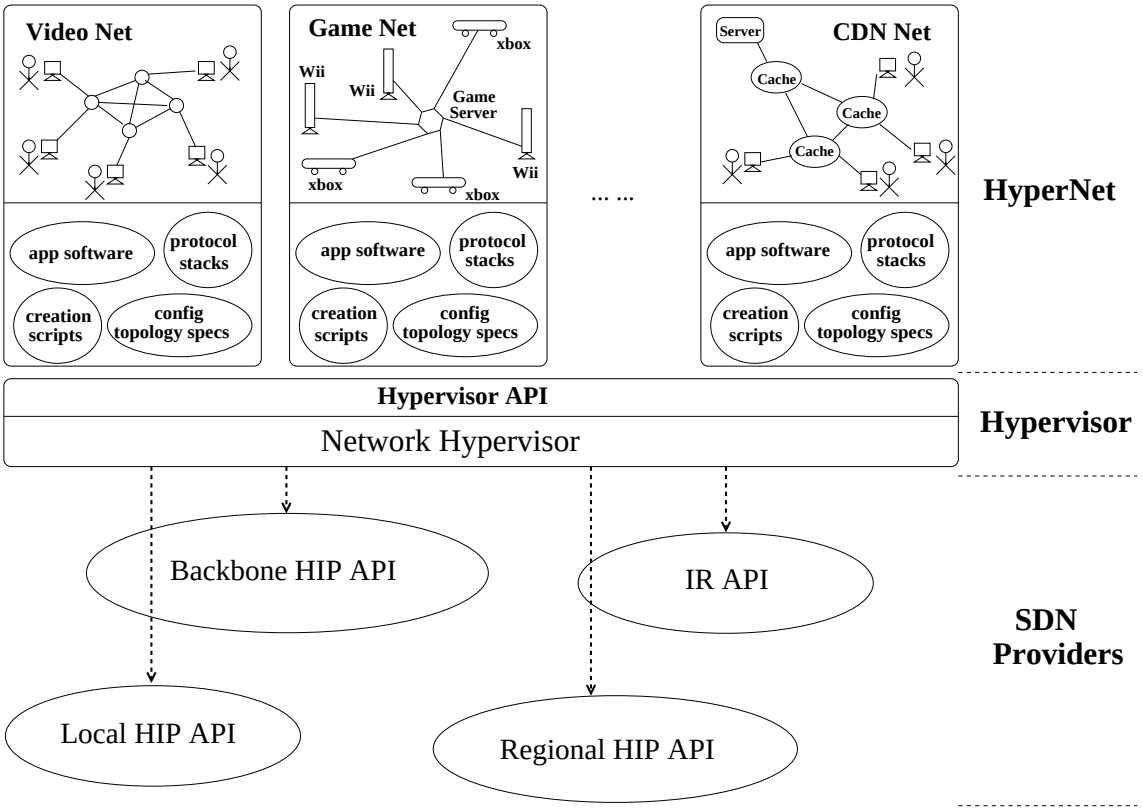


Figure 4.1: The HyperNet Architecture

Fig. 4.1 illustrates the HyperNet architecture. At the heart of the architecture is the *Network Hypervisor*, which performs a role similar to that of a hypervisor for virtual machines. The Network Hypervisor is the platform on which HyperNet Packages run. The Network Hypervisor is responsible for loading and executing the HyperNet Package. The "program" in the HyperNet Package includes the logic needed to obtain network resources from virtual network infrastructure providers, connect resources together to form the topology required by the HyperNet Package,

load the necessary software and/or configuration files on each node of the topology, and then monitor and adapt the topology over time as network conditions change and network participants come and go.

Below the Network Hypervisor reside the virtual network infrastructure providers (VNIPs) described earlier, which provide the resources needed by the HyperNet Package. The Network Hypervisor serves as a common interface to the various VNIPs, which typically have their own particular interface/API.

Running on the Network Hypervisor are any number of HyperNet Packages, each representing a special-purpose virtual network. Each HyperNet Package contains all the application-specific software, configuration files, topology specification code, and monitor code needed to create the virtual network and monitor it as it runs. HyperNet Packages interact with the Network Hypervisor via the Network Hypervisor API. The Network Hypervisor API attempts to hide the details of the underlying VNIPs from the HyperNet Package, allowing the HyperNet Package to be written independent of the specifics of any particular VNIP. In addition the Network Hypervisor monitors the underlying VNIPs and provides "upcalls" to the HyperNet Package (akin to interrupts in an operating system) so the HyperNet Package can adapt to changes in network characteristics or network membership.

Once a HyperNet Package is deployed and running, participants can join or leave the virtual network at any time. Any such membership changes are reported to the HyperNet Package via an upcall from the Network Hypervisor. In this sense, the Network Hypervisor serves as the rendezvous point for all virtual network changes/modifications. All virtual network instantiation/teardown requests from the HyperNet Packages and all participant join/leave requests are handled by the Network Hypervisor. As a result the Network Hypervisor service must be able to scale to handle the potentially large number of requests that it may need to service. Fortunately, the Network Hypervisor is highly parallelizable, allowing us to use cloud-like services to

expand or contract the capacity of the Network Hypervisor.

Although Fig. 4.1 shows a single Network Hypervisor, the architecture allows any number of Network Hypervisors (owned and operated by different entities) to coexist. While all Network Hypervisors offer the same API to HyperNet Packages, they differ in the business relationship they form. Each Network Hypervisor establishes business relationships with the VNIPs it interacts with. It also creates business relationships with virtual network creators and virtual network participants so it can charge for the resources they use. Instead of the users (virtual network creators and virtual network participants) purchasing services directly from VNIPs (which would significantly increase the management overhead for VNIPs), the Network Hypervisor acts as a broker between the network's users and the VNIPs. The Network Hypervisor purchases services in bulk or on demand from the VNIPs with which it has business relationships. It then resells the services to its users (virtual network creators and virtual network participants), thereby avoiding the need for the virtual network creator and every virtual network participant to develop a business relationship with every VNIP. Consequently, Network Hypervisors can compete by arranging different business relationships with the same set of VNIPs. Although we expect the Network Hypervisor to offer these type of business models and broker functionalities — the economics and transaction processing needed to support this functionality is beyond the scope of this thesis and is left as future work.

4.4 The HyperNet Usage Model

Consider the video conference HyperNet Package described earlier in section 1.4. Imagine that end user A wants to have a video conference with end user B. User A begins by downloading a video conferencing HyperNet Package from the HyperNet market. The HyperNet Package is in the form of an executable application. User A then executes the HyperNet Package and when prompted, enters A's and B's IP

addresses to indicate they will be participants in the video network (A and B's IP address could also be specified in a config file to avoid the prompts). The HyperNet Package then creates a virtual network specifically designed for video conferencing and tailored to the participants of this conference, in this case, end users A and B. As described earlier, the HyperNet Package might be designed for MPEG video and will load code onto routers that gives preference to packets containing I frames over those containing P and B frames. It may enable router features that use congestion marking techniques to provide explicit feedback to endpoints so that the video transmission can be adjusted by end applications. Moreover, because it is designed for the network connection only between A and B, the virtual network topology would be specifically designed to satisfy certain video quality. For example, multiple source-routed paths between A and B could be simultaneously used to increase the overall throughput. The HyperNet Package could also include code to track the virtual network usage for each individual participant to charge accordingly, or it may simply pass along all resource usage to the conference organizer user A. In short, the result is a video conferencing virtual network that is tailored to process specific codecs and is optimized for the participants A and B.

After the video conference network is deployed, B can join the network using information about the virtual network, including the name of the virtual network, and any credentials that B needs in order to legitimately join (e.g., a unique participant ID or a password). B also needs to download the corresponding HyperNet End System package (which contains the video conferencing software) in order to send and receive messages on the new network.

The key point here is that any user can easily "deploy" a special-purpose HyperNet Package and start to use the tailored virtual network right away. The HyperNet Package – being a self-contained implementation of the video network – does all the rest. As a result, even non-expert users can deploy rather complicated special-purpose

networks.

4.5 HyperNet Roles

There are six roles involved in building and using a HyperNet Package. Specifically, a HyperNet system consists of *Virtual Network Infrastructure Providers (VNIPs)*, *Network Hypervisor Providers*, *HyperNet Builders*, *Network Creators*, *HyperNet Participants*, and the *HyperNet Marketplace*. The relationship among those roles is illustrated in in Figure 4.2.

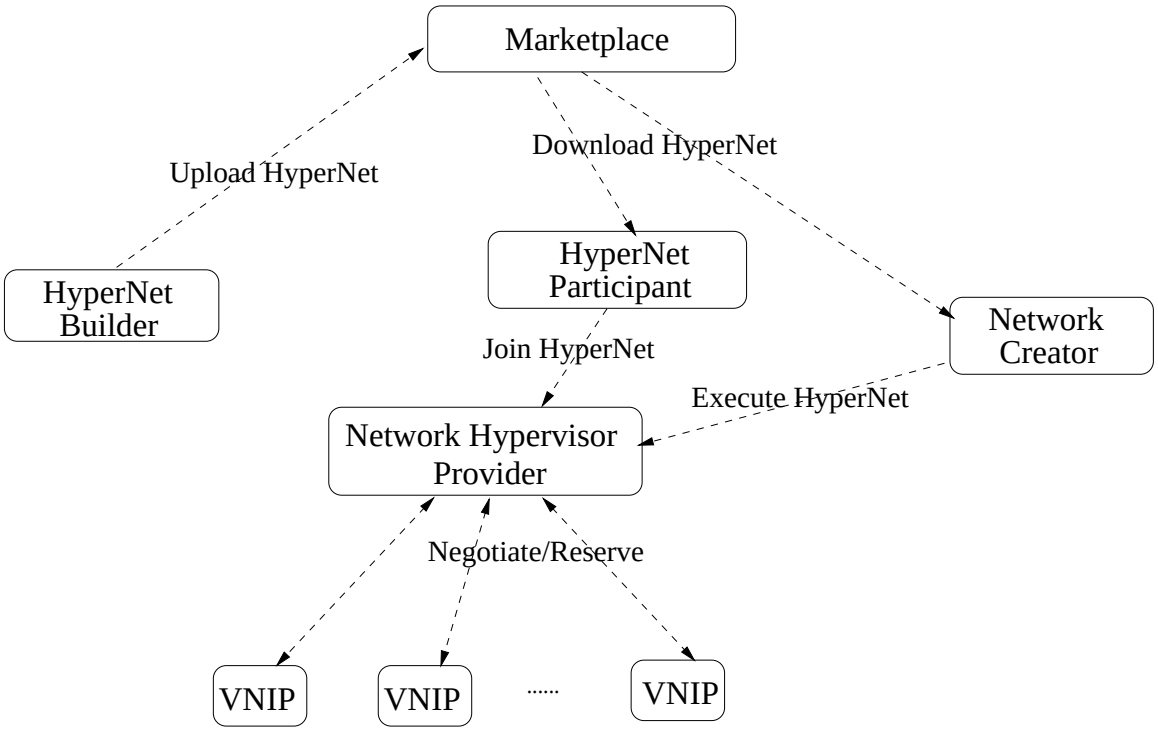


Figure 4.2: Relationship among Roles in a HyperNet Environment

A *HyperNet Marketplace* is a rendezvous point where HyperNet Builders trade their HyperNet Packages with Network Creators and HyperNet Participants, much the same as today's App store concept.

The *Network Creator* is the entity that downloads a HyperNet Package from the market and then runs it on the Network Hypervisor. Because all the expertise that is needed to create a virtual network is encapsulated in the HyperNet Package, a

network creator can be an average user. A network creator does not need to be a *HyperNet Participant* in the virtual network it is creating, but it can be if it wants to.

A *HyperNet Participant* is an end system that joins and uses the virtual network. Just like a network creator, a participant must also obtain (purchase/download) a copy of the HyperNet Package from the HyperNet marketplace and execute it. The only difference is that a participant only executes the end system part of a HyperNet Package, which we call a "HyperNet End System Package". A HyperNet End System package helps a participant join the virtual network and it contains software needed to communicate across the virtual network.

The *HyperNet Builder* is the individual or company with the expertise to write/develop a HyperNet Package. Much like there are "IPhone app developers" for IPhones, there will be HyperNet Builders that design and sell HyperNet Packages. They are network experts and are well-versed with the Network Hypervisor API calls. A HyperNet Package contains two parts: the HyperNet router part and the HyperNet end system part. The HyperNet router part[2] is downloaded and run by the Network Creator to create the virtual network and initialize the routers that comprise the network. The HyperNet end system part (that we call a HyperNet End System package) is downloaded and run by a HyperNet participant in order to join and make use of the virtual network. HyperNet Builders upload their HyperNet packages to the *HyperNet Marketplace* where those packages can be downloaded by *Network Creators* and *HyperNet Participants*.

The bottom layer of the system are the *Virtual Network Infrastructure Providers (VNIPs)*, which provide the (virtual) resources that will be used to construct the virtual network. The Network Hypervisor is the entity that executes HyperNet Packages and creates the virtual networks. We expect Network Hypervisors to be

[2]In the rest of the thesis, if not specifically specified, a HyperNet Package means the router part of the HyperNet Package.

offered as a service by one or more *Network Hypervisor Providers (NHPs)*. Although the architecture allows multiple Hypervisor providers to coexist, all Hypervisors must support the same Network Hypervisor API. NHPs can still differ from each other by establishing different business relationships with different VNIPs.

Gregor et, al. [80] proposed similar ideas for a "virtual network world", but they assumed that ISPs, rather than average users would create and operate the virtual networks. In particular, the authors defined the following business roles: Physical Infrastructure Provider (PIP), Virtual Network Provider (VNP), Virtual Network Operator (VNO), and Service Provider (SP). The PIP owns and manages the physical infrastructure (the substrate) and provides raw bit and processing services which support network virtualization. VNPs are responsible for assembling virtual resources from one or multiple PIPs into a virtual topology. The VNO is responsible for the installation and operation of a VNet over the virtual topology provided by the VNP, and thus provides the tailored connectivity service needed by the SP. The SP is an expert that uses the virtual network as the basis to offer a new network service. This can be a value-added service and then the SP acts as a application service provider, or a transport service with the SP acting as a network service provider. Carapinha et al define similar business roles in their network virtualization model. These papers try to emphasize why and how network virtualization can stimulate network innovation, which directly leads to the business roles necessary in their model. In our design, however, we go one step further: by separating a network service from its provider, we make it possible for anyone to create a specialized network. More precisely, the HyperNet architecture allow us to separate the provider of a network service (i.e., a HyperNet Builder) from the creator of that network service (i.e., a Virtual Network Creator). As a result, even a non-professional user can create and run a new network service. This design creates a potentially huge market for both HyperNet Builders and Network Creators (look at the success of Apple's App store) and also opens the

gate for network innovation — especially for small networks since network creators no longer need to be large service providers who normally offer long-term network services for potentially huge number of users all over the world. Instead, individual network creators can create small networks (both in terms of scale and duration) which last a short period of time and are only for personal use, e.g., a QoS-enhanced video conference network or a low-delay gaming network.

Chapter 5

The Network Hypervisor

The Network Hypervisor provides a unified platform on which HyperNet Packages can "run" to create application or service-specific virtual networks using the resources of the underlying VNIPs. The Network Hypervisor's role is to provide API calls that make it easy for HyperNet Builders to create the desired virtual network topologies. In this chapter, we explore the operations (API calls) that a Network Hypervisor should support to help HyperNet Builders create HyperNet Packages.

We begin by describing the steps needed to build a HyperNet Package. We then discuss the problem of discovering and reserving the VNIP resources needed to build the HyperNet Package. Having identified the tasks a Network Hypervisor needs to do, we present a set of Network Hypervisor API calls that achieve those goals.

5.1 Building a Virtual Network

Creating an application or service-specific virtual network involves a series of steps that the HyperNet Builder wants to control (via calls to the Network Hypervisor). Table 5.1 briefly shows the necessary steps involved in building a virtual network. In the following sections we describe each of the steps, the issues that make each step different, and some possible solutions for implementing each step.

Table 5.1: Steps needed to Build a Virtual Network

Step	Title	Description
1	Identify participants	Specify which participants should be part of the HyperNet network.
2	Identify attachment points	Find the best attachment point for each participant to connect to the HyperNet network.
3	Define the topology	Find the best way to connect attachment points to form a topology for the HyperNet network.
4	Load software on nodes	Load HyperNet-specific software stacks, configuration files, and runtime scripts onto specified programmable nodes in the HyperNet network.
5	Deploy and start the network	Reserve the corresponding programmable nodes and virtual links from the VNIPs and deploy the HyperNet network.
6	Monitor the network	Monitor the usage of the network so as to detect changes and failures, send feedbacks, and charge users.

5.1.1 Step 1: Specify Participants

The purpose of this step is to specify which participants should be part of the HyperNet network. This step may require each participant to send a "join request" message to the hypervisor. This "join" message will allow the hypervisor to verify the participant's identify and record the participant's location information, so that it is later possible for the hypervisor to map this participant to candidate "attachment points" (see Section 5.1.2) that are close to it.

To support this step, the Network Hypervisor needs to offer an interface for participants to join a virtual network (so as to record information such as IP address, end system type, connection type, or platform). It should also be secure enough so that it is hard for a malicious user to illegitimately join the HyperNet network.

To add participants to the virtual network, we designed a *join()* API function call to help a HyperNet Builder to implement the "end system" part of a HyperNet

package, which is used by a HyperNet Participant to join a HyperNet Network. The HyperNet Participant sends a join request to the Network Hypervisor via this API call. Each join request contains a participant ID which is predefined by the Network Creator from a large address space (large enough so that it is hard to forge). The Network Creator gives the participant ID list to the Network Hypervisor and also tells each participant its own participant ID (via offline communication, e.g., email). This participant ID is a per-participant secret that is only known by the participant and the Network Hypervisor. All the Network Hypervisor needs to do is to match the participant ID from a join request with the list from the Network Creator. Alternatively, the Network Creator could simply define one single secret that is going to be used by all participants. The Network Hypervisor's job is then to match the secret in the join request to make sure the joining participant is approved to join. The Network Hypervisor then forwards the information about the participant to the HyperNet Package and lets the HyperNet Package decide how to deal with the participant (e.g., assigning a HyperNet-specific address, a nearby attachment point router, or choosing a tunneling technique to use).

5.1.2 Step 2: Identify Attachment Points

A HyperNet Participant's *Attachment Point* is a programmable node from the VNIP's resource pool. The Attachment Point serves as the entry point for the participant to communicate with a virtual network. Attachment Points are chosen to provide the best possible "entrance" for a participant to communicate with a HyperNet virtual network. Thus, a programmable router that is closest to the participant should be chosen as the attachment point to bring the virtual network as close as possible to the participant.

To find attachment points near each participant, the Network Hypervisor will need a component or service that has the ability to identify the network locations of

participants relative to a VNIP's PR resources (i.e., it must find the network distance between a participant and potential attachment point candidates). Moreover, the service must be scalable enough to handle large numbers of participants scattered all over the world.

Similar problems have been studied in the past, but in a slightly different context. For example, in today's Content Distribution Networks (CDNs), the content providers want to find the best content caches to serve subscribers. The content caches should be both close enough to the content subscribers to reduce network latency and powerful enough to quickly satisfy requests from subscribers. The solution used by many CDNs is to play Domain Name System (DNS) tricks. Upon receiving a DNS request for a web page, the local DNS server redirects the request to a CDN mapping server (the IP address of the CDN mapping server is pre-configured into the local DNS server by the CDN provider). Knowing information about the location of local DNS servers, the CDN mapping server can approximate an end-user's network location by assuming the user is near the DNS server from which the request arrived. The CDN mapping server then chooses a nearby available caching server that is capable of responding to the request. Thus, a CDN network provider is able to choose the best content caches for its customers.

Our Network Hypervisor implementation could take a similar approach to solve this problem in the context of the Network Hypervisor. In this case, the Network Hypervisor would first need to find the local DNS server for the participant (via DNS reverse lookup using the participant's IP address). Knowing the participant's local DNS server, the hypervisor could then choose the programmable router that is closest to the local DNS server as the "entry point" for the participant. The problem is that the Network Hypervisor needs to maintain a mapping from local DNS servers to nearby programmable nodes. However, creating such a mapping requires network location information about both programmable nodes and DNS servers (and, like the

CDN solution, this solution relies on cooperation with the DNS system).

An alternative way is for the Network Hypervisor to delegate this mapping task to the corresponding underlying VNIPs. In this way, the Network Hypervisor would pass the request on to the VNIPs and let the VNIPs determine whether they have programmable routers that are near the participant. Because two VNIPs may both be close to a particular participant, the (estimated) distance from the participant to the nearest programmable router must also be provided to help the hypervisor select the appropriate programmable router. Yet another alternative is to create a third-party service that is responsible for learning the location of VNIP resources and being able to find resources near participants. For our prototype implementation we took this approach, implementing a third party service which we describe in Section 6.2.

5.1.3 Step 3: Define the Topology

Given the list of VNIP resources, the HyperNet Builder must create an appropriate virtual network topology. Determining the best way to interconnect the VNIP resources is a significant challenge. The topology of the virtual network should be designed based on the available resources, the functionality of the virtual network, and the participating end systems.

Section 5.2 examines this problem of resource discovery and topology specification in detail. We developed a set of Network Hypervisor APIs for a HyperNet Builder to use to explore the underlying VNIPs' physical topology, as well as to accomplish tasks such as finding the shortest path between PRs, and finding a central point among a set of PRs. In addition, we also built a HyperNet Topology Library to make it easy for a HyperNet Builder to build a topology of a certain shape, e.g., a shared tree, a source-specific tree, a ring, a star, or a mesh network. The list of API calls in the HyperNet Topology Library can be extended by HyperNet Builders to add more functionality to explore the underlying topology in the future. In short, the basic

set of Network Hypervisor APIs we provide (described in Section 5.3) is fine-grained enough for a HyperNet Builder to create a topology of any shape.

5.1.4 Step 4: Load Software on Nodes

After the topology of a virtual network is defined, the HyperNet Builder needs to load specific software packages, configuration files, and libraries onto specific programmable nodes in the topology. A mechanism needs to be designed to load application-specific code onto VNIP nodes and on participants' end systems.

We assume that the HyperNet Builder, as an experienced programmer, knows which network code (including the protocol stacks, application-specific code and any services that runs on a VNIP node) should be used in which circumstances. For example, in the case of a High-Definition video broadcasting HyperNet network, when a participant with a small screen cell phone connects to the virtual network via a wireless link, a special appliance should be loaded on the attachment point PR which connects to the participant to deal with video compression and high loss rate. As a result, the HyperNet Package (designed by the HyperNet Builder) should be able to load any pre-defined and pre-configured software packages (e.g., virtual appliances) onto any reserved programmable node. We developed an API call "loadApp()" (a more detailed description will be given in Section 5.3) for a HyperNet Builder to load and execute any application (e.g., code, scripts, and configurations) onto any reserved programmable nodes.

HyperNet Participants do not use "loadApp()" to load appliances. Instead, we ask a HyperNet Participant to download and use a HyperNet End System package, which includes all the software necessary for a participant to join and use a HyperNet virtual network.

5.1.5 Step 5: Deploy and Start the Virtual Network

This step involves reserving the specified VNIP resources and setting up virtual links among them. Once the virtual network has been defined (say via the RSpec in GENI), it simply needs to be specified to the VNIP to be deployed. If the virtual network includes programmable nodes in multiple VNIPs, then the network specification (i.e., an RSpec file in GENI) needs to be provided to all involved VNIPs. The programmable nodes needed to create inter-VNIP virtual links must also be identified in the network specification so that special virtual links (e.g., a GRE tunnel) could be created connecting programmable nodes in different VNIPs.

5.1.6 Step 6: Monitor the HyperNet network

HyperNet Monitoring is necessary to (1) detect and react to changes in participants, (2) give feedback to the Network Creators so that the Network Creators are aware of the resource utilization, node health, and participant status[1], (3) catch errors happening in the virtual network so that proper actions can be taken in time and (4) keep a record of the resources used for billing purposes. Since network monitoring is out of the scope of this thesis, we omit the analysis of the challenges in this step. However, we can imagine that with the help of network monitoring, the Network Hypervisor will be able to not only find the best paths/topology based on static information (e.g., number of hops), but also based on real-time dynamic information (e.g., live traffic graphs). Today, network monitoring is already happening in ISPs [81]. We expect future VNIPs will also maintain their own monitoring infrastructure and report that information to the Network Hypervisor.

[1]Feedback is also useful to HyperNet Builders to debug the HyperNet Packages they are creating.

5.2 Discovering/Defining the Topology

Each HyperNet Package will only use a subset of the resources available from VNIPs. The question is "how does a HyperNet Package discover and define/specify the set of resources it wants to include in its virtual network?" In general, HyperNet Packages have little desire to "see" all the possible resources a VNIP has (and VNIPs may not want to "show" all of their resources and connections). Instead, HyperNet Packages typically want to know as little as possible about the underlying VNIP resources and the VNIP physical topology. Only in cases where the details make a big difference does the HyperNet Package want to "drill down" to the details of the VNIP's resources/physical topology.

To accommodate multiple levels of detail, the Network Hypervisor begins by providing the HyperNet Builder with a high-level, abstract, view of only the VNIP resources that are immediately needed by the HyperNet Package, namely, Attachment Points for the HyperNet Participants (the HyperNet Participant list can be pre-defined by a Network Creator via, e.g., a configuration file, which we will describe later). The Network Hypervisor then uses the concept of a *Transparent Tunnels* to connect attachment points together. The key feature of a Transparent Tunnel is the ability to "drill down" to see the details (i.e., the resources) used to create the tunnel, and to modify or adjust those resources if greater control is desired. Typically, HyperNet Packages will view the VNIP topology at any of three levels of detail, as described in the following sections.

5.2.1 Attachment Point View

Fig. 5.1 shows a topology in which only Attachment Points in the VNIP are visible. Under this view, the VNIP only finds and reveals the nearest PR to every participant. We call the links connecting Attachment Points "Transparent Tunnels". If a VNIP controls all the PRs and WPs along a transparent tunnel along the path, then it

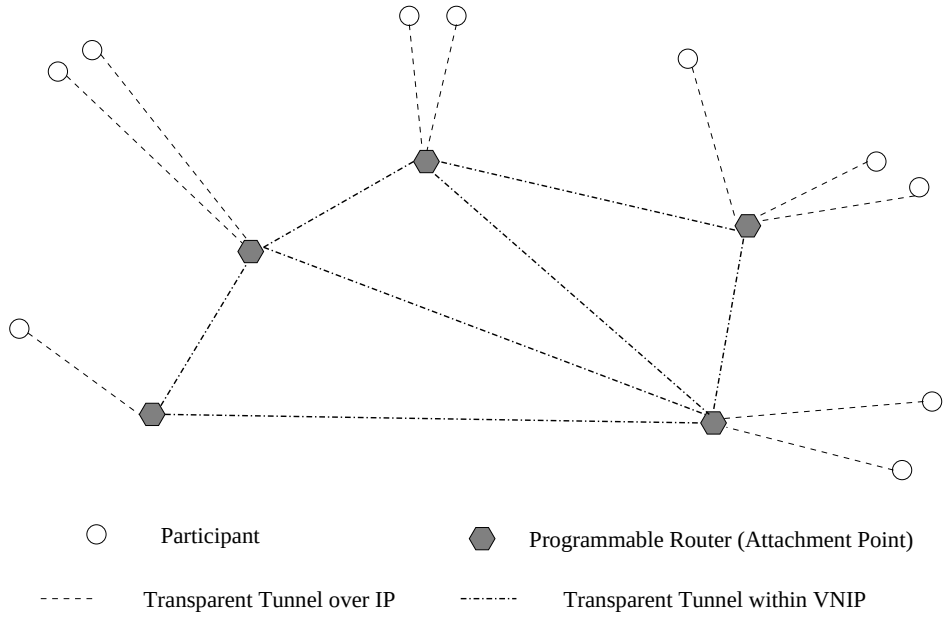


Figure 5.1: Attachment Point View

is possible for the HyperNet Builder to provision the capacity of the transparent tunnel. On the other hand, a Transparent Tunnel may traverse the existing Internet (i.e., IP routers) where the VNIP has little control. Examples include GRE tunnels between participants and their attachment points. In this case, a transparent tunnel only provides a means of connectivity to the HyperNet Participants. VNIPs have no control over (and often no information about) the QoS of such tunnels.

5.2.2 Key Resource View

For those HyperNet Builders who want to reserve certain types of programmable routers/servers and/or control the network topology, the Network Hypervisor provides a second level view of its topology and resources. Fig. 5.2 illustrates an example network in which the HyperNet Package wants to utilize a programmable server, centrally located between the attachment points and near a PR of its own. In this view, the HyperNet Package asks to see resources that meet certain requirements and then selects some subsets and maps them into the topology using Transparent Tunnels. Again, transparent tunnels may contain hidden PRs, Way Points, and

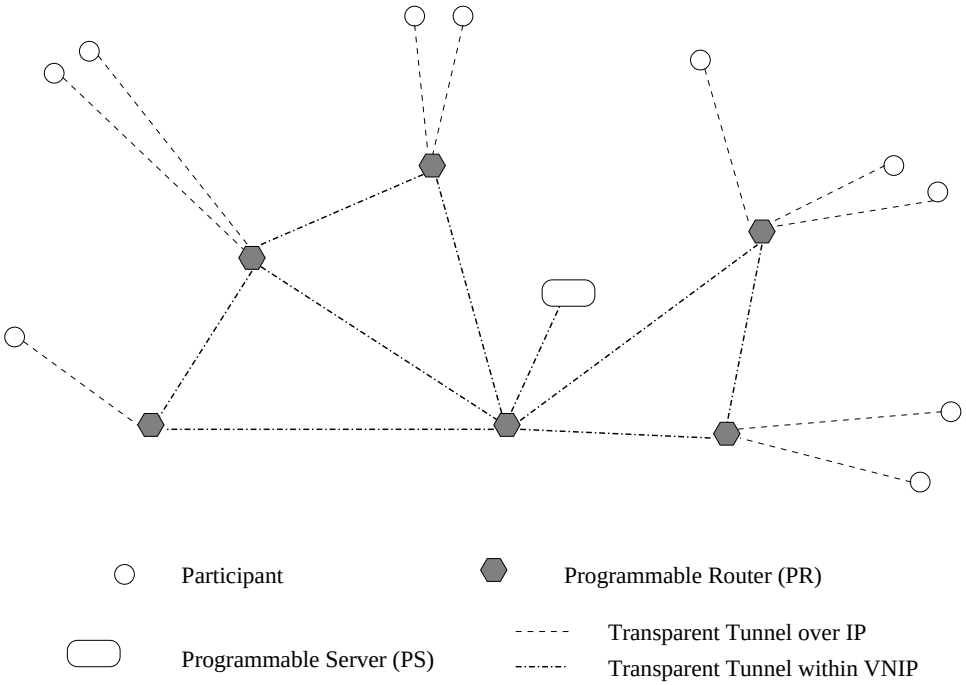


Figure 5.2: Key Resource View

IP-based routers within them. Consequently, it is possible for multiple transparent tunnels to traverse a single physical link. In this model, a HyperNet Builder only needs to reserve and control a small set of strategically selected programmable routers and servers, as well as the transparent tunnels connecting them. All the path selection and traffic engineering tasks within the transparent tunnels are left to the VNIP. The programming tasks for HyperNet Builders under this model are simplified since HyperNet Builders only need to pick and program key programmable routers and servers with application-specific network functionality. Again, the HyperNet Builder can design the HyperNet Package such that it is required that a Network Creator defines an initial participant list to use this HyperNet package. Thus, the HyperNet Package can initialize the HyperNet network by picking the best attachment point PRs and the best key PRs specifically for the pre-defined participants.

We expect that Attachment Point View and Key Resource View will be used by most of the "entry level" HyperNet Builders.

5.2.3 Detailed Topology View

For those advanced HyperNet Builders who desire to choose the paths in a transparent tunnel themselves, the Network Hypervisor offers a third view of its resources. Fig. 5.3 illustrates this *Detailed Topology View*. In this view, the transparent tunnels become

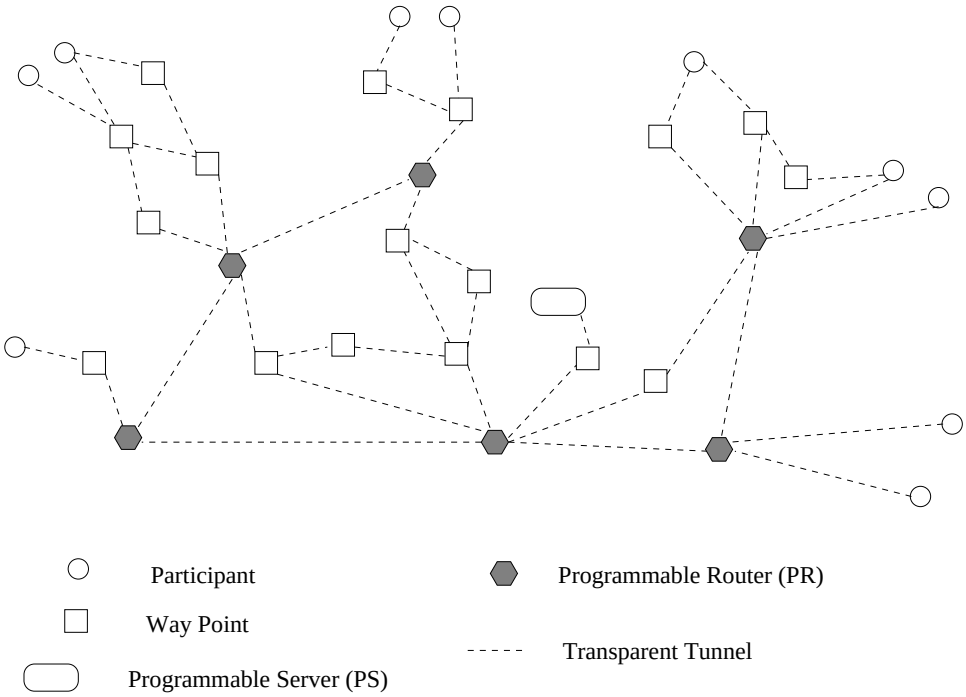


Figure 5.3: Infrastructure Provider's view of its managed networks III

visible and the HyperNet Builder can discover the PRs and Way Points a transparent tunnel contains. A transparent tunnel may actually "hide" an entire *network of PRs and Way Points*, which become visible in this view. In other words, a tunnel is not just a sequence/path of PRs and WPs but rather is an entire network. By exploiting the "hidden" details within transparent tunnels, a HyperNet Builder is offered greater control over the resulting virtual network. In this case, the Network Hypervisor provides APIs to (1) expose transparent tunnels and (2) modify the set of PRs/WPs used in a tunnel (i.e., to change the path used by a tunnel). We expect that this model will be used mostly by advanced HyperNet Builders.

Not only can transparent tunnels between programmable routers and programmable servers be customized, transparent tunnels between participants and their attachment point PRs can also be customized. Of course, this customization requires that the network devices between the participant and its attachment point routers be controlled by the VNIP.

5.3 The Design of a Network Hypervisor

The *Network Hypervisor* is a key component of the HyperNet architecture. It is the glue that connects the HyperNet Packages with Virtual Network Infrastructure Providers. The Network Hypervisor is responsible for executing HyperNet Package, obtaining network resources from VNIPs, connecting resources to form the topology required by each HyperNet Package, helping to load the necessary software and configuration files on each node of the topology, and monitoring and adapting the topology over time as network conditions change and HyperNet Participants come and go. Fig. 5.4 illustrates the Network Hypervisor in relationship to HyperNet Packages, the HyperNet Library used by HyperNet Packages, and VNIPs.

The bottom half of the Network Hypervisor contains different VNIP handlers that the hypervisor uses to talk with different VNIPs. The top half contains a set of *Hypervisor API calls* that a HyperNet Package can invoke to (1) select, customize and reserve (programmable) nodes and links, (2) explore the physical topology and build virtual networks of certain shapes, (3) register, deploy and tear down HyperNet virtual networks, and (4) communicate with end systems to accomplish joining and leaving networks, sending and receiving packets. Building on the Network Hypervisor API calls, we implemented a set of HyperNet Library calls that offer a higher-level programming interface to HyperNet Builders. Example library calls include building a tree topology, loading non-standard routing protocols (e.g., multicast

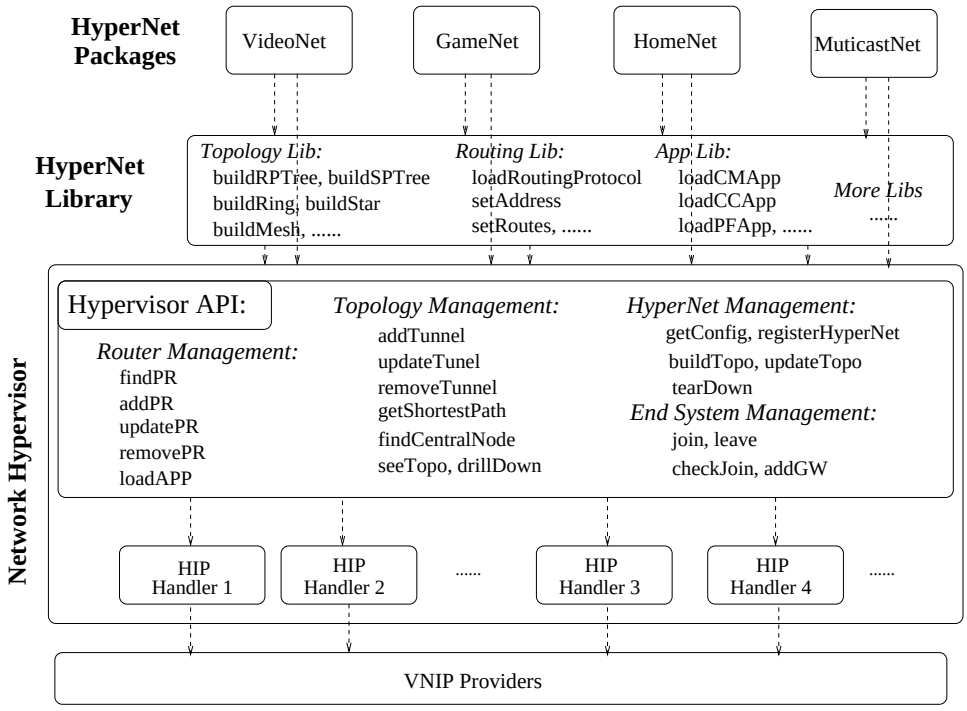


Figure 5.4: Network Hypervisor API Layers

routing protocols), setting up static routing tables for a user-defined path, and loading special-purpose networking applications onto nodes/topologies.

In the following we take a closer look at the Network Hypervisor and the API calls it offers.

5.3.1 VNIP Handlers

A VNIP handler is the interface that the Network Hypervisor uses to talk to an underlying VNIP. As mentioned earlier, we envision the existence of multiple VNIP providers in the future Internet. The Network Hypervisor needs to be able to handle the various protocols offered by each individual VNIP in order to communicate with them.

A VNIP handler's job includes: (1) connecting Network Hypervisor API calls into the appropriate VNIP call (e.g., reserve/remove/update a node/link, load a file onto a node, etc.), (2) fetching the topology/resource-availability information from the

VNIP so that the Network Hypervisor maintains the topology information (more on this in chapter 6), and (3) retrieving network usage statistics and monitoring data from the VNIP.

5.3.2 Network Hypervisor API Calls

The Network Hypervisor API calls are designed to give HyperNet Builders the "programing" tools/environment they need to programmatically create and deploy application-specific virtual networks. The API calls are roughly divided into four classes: (1) Router Management, (2) Topology Management, (3) HyperNet Management and (4) End System Management. Figure 5.4 shows the API calls associated with each of the classes.

Router Management calls are used to find and add programmable routers into the topology. Router Management API calls include:

node *findPR*(Participant *p*, Restriction *r*)

findPR finds the nearest attachment point for for participant p, satisfying restriction r.

The Network Hypervisor first selects a small set of VNIPs that are thought to be close to participant p. In our current Internet, a participant p can be identified by its IP address. A VNIP can compute/keep a list of its best PRs for every possible IP network, much like a CDN provider keeps a list of its best caches for every DNS server. Each of the selected VNIPs then sends back to the hypervisor its recommended attachment point, as well as the expected network performance between the participant and the attachment point. A lightweight probing mechanism can be used by each VNIP to generate the performance result (e.g., ping for RTT, traceroute for number of hops, etc.). The probe messages will be generated from each VNIP towards the participant. The performance results will be collected by each VNIP and then reported to

the hypervisor. The hypervisor examines the returned probing statistics and chooses the attachment point with the best performance (i.e., the closest PR towards the participant). The restriction parameter r could be used to specify the desired capacity of the attachment point, such as the CPU, memory/disk space, and the number of available network interfaces on the attachment point.

Node *addPR*(Node *pr*)

This function reserves a programmable router *pr*. The data structure "Node" allows the caller to specify the type of the programmable router (PC or VM) and the capacities of the PR (CPU cores, memory size, disk space, etc.). If the virtual network is not yet deployed, the instantiation of this PR is postponed until the user calls *buildTopo()*. If the virtual network is already deployed, the instantiation of this new PR is postponed until the user calls *updateTopo()*. This API call will fail if the hypervisor does not think that a PR with the specified capacity is available from the VNIP. Errors/failures might also happen at the time of deployment when the VNIP tries to reserve the PR for the HyperNet Package, which will result in an error message returned in the *buildTopo()* call or in the *updateTopo()* call.

The current *addPR()* API call does not consider business models but we can imagine that this function call is the primary method by which a HyperNet virtual Network Creator pays for resources. As a result, one might consider adding other parameters (e.g., credentials) to this function call in the future to support various business transactions.

Node *updatePR*(Node *pr*)

This function updates the characteristics/capacity of a programmable router *pr* that is already in the network. The Network Hypervisor identifies an existing PR via the "virtual_id" field of that *pr*. Again, the actual effect is

deferred until the user actually deploys or updates the network via *buildTopo()* or *updateTopo()*. This API call will fail if the hypervisor thinks that the newly requested characteristics are unavailable.

Node *removePR*(Node *pr*)

This function removes a programmable router *pr* from the network, along with all virtual links that connect *pr*.

Boolean *loadApp*(Node *pr*, Application *app*)

This function loads an application *app* onto a programmable router node *pr*. The *app* may be a self-extracting software package or a virtual appliance. *loadApp()* can be used both before a virtual network is deployed and after a virtual network is deployed. In the first case, *app* is simply "registered" with the Network Hypervisor. Only after the network is deployed is the application uploaded onto the corresponding PRs and executed. In the latter case, the Network Hypervisor immediately uploads and executes the application on the corresponding PR[2].

The HyperNet Package might want to load additional software onto existing programmable routers after the HyperNet network is deployed and running. Sometimes the HyperNet Package will not know how to configure the routers until the network is up and running and participants have joined. For example, when a participant has joined, a new (IP) address is assigned to the corresponding HyperNet Participant as well as the new (GRE) interface on the chosen gateway router. In order that all other routers in the virtual network to know how to route to the newly joined participant, proper routing table entries need to be added (unless there is a dynamic routing protocol running in the

[2]We assume that the Network Hypervisor has the necessary privileges, credentials, ssh keys, etc to copy software onto programmable nodes and execute it. Network Creators might need to provide this information when they first create a HyperNet network, or the Network Hypervisor might use its own credential to allocate/reserve/control resources.

virtual network). This configuration cannot be done when the network is first deployed.

HyperNet Participants are responsible of finding, downloading, and executing the corresponding HyperNet End System package to join and use the HyperNet network. The HyperNet End System package includes all the software that is necessary to run on the HyperNet Participants' end systems in order to use the HyperNet network. Hence, software that runs on end systems is not loaded using *loadApp()* API call. Instead, it is included in the HyperNet End System package, which runs on a HyperNet Participant's end system.

API calls related to *Topology Management* include:

Link *addTunnel*(Node *pr1*, Node *pr2*, NodeSet *way_points*, Restriction r)

This function reserves a Transparent Tunnel between two nodes $pr1$ and $pr2$ that travels through a set of specified way points or PRs, satisfying restriction r.

This call results in all the way points specified being configured with proper forwarding table entries (or VLAN configurations) that forward packets from $pr1$ to $pr2$ following the node order defined in NodeSet way_points. Because a way point is a restricted form of a programmable router, the list of way points can contain programmable routers. when this occurs, the PR will simply be treated as if it were a WP. When way_points is null, it returns a transparent tunnel connecting $pr1$ and $pr2$ and follows the default path provided by the underlying VNIP (typically the shortest path). Restriction specifies the capacity of the transparent tunnel, such as its bandwidth, delay and loss rate. This call will fail if $pr1$ and $pr2$ are not already included in the topology. If the underlying VNIP decides that the required restriction can not be met, then an

error will be returned at the time of deployment (e.g., returned by *buildTopo()* or *updateTopo()*).

Link *updateTunnel*(Link *myTunnel*)

This function updates the properties of an already reserved tunnel *myTunnel*. Properties of a "Link" structure include the tunnel_id defined by the two end points of the tunnel, bandwidth, latency, and packet loss rate.

Link *removeTunnel*(Link *myTunnel*)

This function removes an existing tunnel *myTunnel* from the virtual network. A tunnel will no longer be available to forward packets after being removed from the virtual network.

Path *getShortestPath*(Node *a*, Node *b*)

This API call finds the path with the smallest hop count between Programmable Routers *a* and *b*. The returned Path is an ordered list of programmable nodes. All nodes in the returned path are Programmable Routers. The virtual links in the returned path connecting the returned PRs are all transparent tunnels.

Node *findCentralNode*(NodeSet *nodes*)

This API call enables a HyperNet Builder to find a PR that is centrally located in the middle of a group of PRs. The "Central Node" is defined as the Programmable Router at the center of the given PRs. "Center" might mean that it provides the minimum number of network hops to all given PRs, or it might mean that it provides the minimum average RTT to all given PRs. This function is very useful in cases where a centralized server is needed, or a Rendezvous Point (say for a multicast tree) is needed. For example, to create a virtual network for a real-time online first person shooter game where each of the player wants low delay to the game server, a centralized node needs to be chosen.

Graph *seeTopo*()

This API call gives HyperNet Builders the ability to see the entire topology available from all VNIP providers. This call returns both the topology information as well as the resource availability information (e.g., whether a programmable node is already reserved; or the "weight" value of a link indicating how congested that link is). Instead of giving partial topology information (as in *getShortestPath()* and *findCentralNode()*), this API call returns the entire topology available from all VNIPs. This advanced API call gives HyperNet Builders more information about the underlying VNIP topology, but it introduces more complexity in deciding which programmable nodes to reserve and which tunnels to create. This API call needs support from the underlying VNIP and the underlying VNIPs can control/limit what gets returned (VNIPs may decide to hide part of their physical networks from the user for security reasons, much like ISPs do today).

PathSet *drillDown*(Link *myTunnel*)

This function explores path choices within the transparent tunnel *myTunnel*. Each returned path contains a number of way points (or PRs) contained in the transparent tunnel. This is the main function for a HyperNet Builder to "drill down" a transparent tunnel to explore the potential path choices. Depending on the VNIP's policy, it may not reveal all the path choices to the user. Moreover, if *myTunnel* only contains a directly connected physical link or it traverses through a set of IP routers over which the VNIP has no control, this API call returns null.

The previous calls dealt with the definition of the topology and allocation of resources within a HyperNet. The following API calls deal with the management of HyperNets such as registering a HyperNet with the Network Hypervisor:

Config *getConfig*(String *configFile*)

This API call is used by a HyperNet Package to read the content from a HyperNet configuration file and return the configuration. Argument $configFile$ is the path to the HyperNet configuration file. A HyperNet configuration file is the mechanism by which a Network Creator specifies features of the network to be created. It includes information related to the HyperNet network, such as the name of the HyperNet network, necessary credentials/secrets, and a pre-defined participant list. A HyperNet configuration file is filled solely by a Network Creator (A configuration file template is typically provided within a HyperNet Package to facilitate a Network Creator to accomplish this task). However, the HyperNet Builder, being the designer of the HyperNet Package, defines the syntax and semantics of the configuration file.

APIMessage *registerHyperNet*(Config *myConfig*)

This API call registers a HyperNet network using the configuration specified in $myConfig$. As a result, multiple Network Creators can download and execute the same HyperNet Package, provide different configuration files while executing the HyperNet Package to create different HyperNet networks. Thus, different HyperNet networks (with different names specified in the configuration file) will be registered using this API call. If a HyperNet network with the same name is already registered in the Network Hypervisor, then this call will fail and return an error message. Upon successful registration, the Network Hypervisor checks the credentials contained in $myConfig$ and then registers/saves the other configuration information from $myConfig$ such as the participant list and "join secret"[3].

APIMessage *buildTopo*(String *HyperNetName*)

[3]A secret is set by a network creator who does not know the identities of its HyperNet participant, but wants its HyperNet network to be accessible by only those who know the secret

This function deploys the virtual network associated with *HyperNetName*. The *HyperNetName* is defined by the Network Creator in the configuration file. The API call reads the virtual network description file (e.g., the RSpec file in protoGENI) generated by the Network Hypervisor using the previously described API calls and then deploys the virtual network using the VNIP API. If the underlying VNIPs are not able to fulfill the deployment task (e.g., there is not enough resources or some internal error occurred), this API call will fail and return the detailed error message to the caller.

APIMessage *tearDown*(String *HyperNetName*)

This API call tears down a virtual network. It clears all the state information related to the virtual network in both the Network Hypervisor and all the programmable nodes reserved from the underlying VNIP. All resources including PRs, WPs and any virtual links reserved are released and marked "available" in the VNIP's resource pool.

The previous calls dealt with managing and defining a HyperNet network. Those API calls are called from the network part of a HyperNet Package. The following API calls deal with the management of a HyperNet End System, and thus are called from the end system part of a HyperNet Package:

TunnelInfo *join*(String *HyperNetName*, Credential *credential*, Info *myInfo*)

This API call is used by a potential HyperNet Participant to join an existing HyperNet network. The HyperNet Participant uses this API call via a HyperNet package running on the participant's end system. In particular, every HyperNet Package contains a section of code designed to run on participant machines. We call this the End System Package, and participants obtain the End System Package by download the HyperNet Package and extracting the End System Package.

This API call sends out a HyperNet join request to the Network Hypervisor and waits for a "TunnelInfo" structure to be returned (TunnelInfo will be returned by the Network Hypervisor upon receiving a corresponding "addGW()" API call described later). TunnelInfo includes information necessary for the end system to set up a tunnel with its assigned attachment point router. The join request message includes: the $HyperNetName$ of the HyperNet network, the (optional) *credential* of the joiner, and (optionally) the participant's identity information $myInfo$. A participant's identity information uniquely identifies a participant within a HyperNet network. An example could be a participantID assigned by the Network Creator. The participant information can further include such things as the participant's IP address, or the participant's end system connection type (wired or wireless). The hypervisor, upon receiving a legitimate join request (via the *checkJoin()* API call described later), in turn assigns a HyperNet-specific address to this participant and creates a tunnel between the participant and the assigned attachment point router. The hypervisor also informs the HyperNet Participant about its assigned HyperNet-specific address and the HyperNet attachment point router so the participant can update its routing table, setting the assigned attachment point router as the default gateway for all packets destined to the HyperNet virtual network.

JoinRequest *checkJoin*(String *HyperNetName*)

This API call is called by the "join request handling code" in the HyperNet Package to handle "join" requests. It waits for join requests made to HyperNet network whose name is $HyperNetName$. Upon receiving a new join request, this API call returns a "JoinRequest" data structure to the caller. The hypervisor essentially "wakes up" the "join request handling code" in the HyperNet Package.

Boolean *addGW*(String *HyperNetName*, TunnelInfo *tunnelInfo*)

This API call is used by the "join request handling code" to assign an attachment point and instruct both the attachment point PR and the joining participant to establish a tunnel between themselves. The "TunnelInfo" data structure includes the information about a joiner (identified by its IP address or participantID), the assigned attachment point for the joiner, and the assigned HyperNet-specific address for the requester. The HyperNet Package's "join request handling code" is in charge of creating the TunnelInfo. Upon receiving an *addGW()* API call, the hypervisor will "wake up" a waiting JoinRequest by matching the information in the TunnelInfo with the JoinRequest, and returning the TunnelInfo to the corresponding *join()* API call.

Boolean *leave*(ID *HyperNetName*)

This API call is used by an end system to leave a HyperNet network. Upon receiving a *leave* request, the Network Hypervisor clears the state information related to the participant from both the Network Hypervisor and the participant's corresponding attachment point PR.

5.3.3 The HyperNet Library

In addition to providing a basic set of API calls to achieve basic node-level and link-level tasks, our HyperNet architecture also provides a *HyperNet Library* which offers advanced topology-level and system-wide functionality that helps HyperNet Builders develop their HyperNet Packages. All HyperNet Library functions are implemented using the existing Network Hypervisor API calls or other HyperNet Library API calls. For example, the *buildRing()* Library call (which helps build a Ring topology) makes use of the *findPR()*, *addPR()* and *addTunnel()* Network Hypervisor API calls to achieve its functionality. The obvious advantage of the HyperNet Library calls is that they make the HyperNet Builder's task of composing a HyperNet Package

a lot easier. The Library does not sit inside the Network Hypervisor but rather is dynamically "linked" with the HyperNet Package. As a result, any third party developer (instead of the Network Hypervisor service provider) can contribute to make the HyperNet Library more useful. Of course, the Network Hypervisor provider can also add more HyperNet Library calls at any time to make it more capable. The current Library offers the following calls:

Topology *buildRing*(NodeSet *nodes*);

This Library call creates a topology connecting all the given nodes into a ring. This call first uses *addPR()* to reserve all the PRs specified in parameter *nodes*. It then reserves transparent tunnels connecting the PRs into a ring using the *addTunnel()* hypervisor API call.

Topology *buildStar*(Node *center*, NodeSet *nodes*);

This Library call creates a star topology having the node *center* as the center of the star. Each of the nodes in the given NodeSet *nodes* then connects to the center with a transparent tunnel, forming a star topology.

Topology *buildMesh*(NodeSet *nodes*, int *degree*);

This Library call creates a mesh topology, connects all the given nodes into a connected topology. Each node on the connected mesh network has at least *degree* number of links connecting it with other nodes. If *degree* is higher than the number of nodes, then this library call will create a fully connected graph, or a "full mesh", as described next.

Topology *buildFullMesh*(NodeSet *nodes*);

This Library call creates a complete graph (or "full mesh") topology connecting the given nodes.

Tree *buildRPTree*(NodeSet *nodes*);

This Library call creates a Rendezvous-Point-Based tree, with all nodes in

NodeSet *nodes* as leaves. The resulting tree is based on a rendezvous point which is selected as the central point among the given NodeSet (via the Network Hypervisor *findCentralNode()* API call). The tree is formed by the shortest paths from the given nodes to the chosen rendezvous point.

Tree *buildSPTree*(NodeSet *nodes*, Node *sender*);

This Library call creates a shortest-path tree, with all nodes in NodeSet *nodes* as leaves. In this case, the links of the tree are formed by all the shortest paths from the given node *sender* to all nodes. When *sender* is given as NULL, any node in the tree can be a sender and the tree is formed by combining all shortest paths between any two nodes in the given NodeSet. Shortest paths are found by using the *getShortestPath()* hypervisor API call.

Boolean *setAddress*(Node *pr*, Interface *intf*, Address *address*);

This Library call is an extension of the *loadApp()* call. It sets the *address* of the interface *intf* on node *pr*. Under Linux, it is achieved using the *ifconfig* program. This library call is created for a HyperNet Builder to manually configure the IP addresses used in his HyperNet Package, in cases where a dynamic address assigning protocol (e.g., DHCP) is not used in the HyperNet Package.

Boolean *setRoute*(Node *pr*, Address *dstAddress*, Node *nextHop*);

This Library call is an extension of the *loadApp()* call. It adds a static routing entry into the routing table of node *pr*. This routing entry defines the next hop node *nextHop* for every packet destined to *dstAddress*. Under Linux, it is achieved by the *route* program.

Boolean *loadRoutingProtocol*(Topology *myTopo*, Protocol *myProto*);

This Library call loads the specified routing protocol *myProto* onto each PR of the given topology. It makes use of the *loadApp()* call to load pre-configured

protocol stacks on to each node in the topology and use those protocol stacks. For example, to load and run a user space PIM multicast protocol, we define a pre-configured pimd [82] routing daemon to be loaded onto each node of *myTopo* via this API call.

Besides the set of HyperNet Library calls we introduced above (which are also implemented in the Network Hypervisor), we can easily imagine many other HyperNet Libraries that can be useful in composing different kinds of HyperNet Packages. For example, an "Application Library" could help the HyperNet Builder load well-configured applications (software stacks such as a Content-Management App, a Congestion-Control Traffic Engineering App, or a Packet-Filtering App). Similarly an "Instrumentation Library" could help the HyperNet Builder load monitoring tools onto critical nodes in the topology to monitor resource utilization, networking performance etc. An "OpenFlow Library" that only deals with OpenFlow networks could help make it easier to, for example, create an OpenFlow controller that does load balancing. A "Redundancy Library" could help the HyperNet Builder create backup servers and backup paths, etc. Clearly the Library list could grow long, but the point is, a third party developer can easily contribute to the HyperNet Library using the existing basic Network Hypervisor APIs.

5.4 Configuring HyperNet Packages

The main contribution of our HyperNet Package abstraction is that it minimizes the effort needed by an average user (Network Creator) to deploy and operate a personal specialized virtual network – the Network Creator only needs to "run" the HyperNet Package. While that is all it takes to "run" some HyperNet Packages, some other HyperNet Packages may need to be further configured by the network creator. In this section, we describe the possible ways in which a Network Creator can further tailor/configure an application-specific HyperNet network.

Network Creators use a HyperNet configuration file to specify configuration information about the virtual network being created. As mentioned earlier, we provide a Network Hypervisor API call *getConfig()* to read the configuration file. Since virtual networks vary from one another, we leave the design to the HyperNet Builders to decide the package-specific parameters in the configuration file that a Network Creator can use to control the virtual network it is creating. As a result, the HyperNet Builder is the expert who designs the semantics of a HyperNet configuration file and the Network Creator is the average user who "fill out" the HyperNet configuration file subject to the semantics defined by the HyperNet Builder. Although the *getConfig()* API call reads and parses the entire config file, there are only a small set of parameters (some are mandatory, some are optional) that a HyperNet Package reads and reports to the Network Hypervisor via the *registerHyperNet()* API call. The small set of configuration parameters includes:

- *HyperNetName* (mandatory): The name of the HyperNet network the Network Creator is creating.
- *UserCredential* (mandatory): The credential that shows one is authorized to run the HyperNet Package and deploy the virtual network through Network Hypervisor onto the VNIP. This field might also include payment information (which is not discussed in this thesis), such as a credit card number.
- *ParticipantsList* (optional): The addresses or host names of the allowed HyperNet Participants. The Network Hypervisor can match the addresses of future join requests with this list to filter out unauthorized participants.
- *Expandable* (optional): If this parameter is set to 1, then the HyperNet Package allows participants not listed in the ParticipantList to join.
- *HypervisorServer* (optional): The address (or domain name) of the specific Network Hypervisor server that the Network Creator wants to use. We envision

the co-existence of multiple Network Hypervisor servers running in parallel in the future. This option allows a Network Creator to choose a specific Network Hypervisor to use. Alternatively, a single Network Hypervisor service provider might use cloud resources to offer an expandable version of the Network Hypervisor, much like the tricks played in today's DNS systems where a single domain name might be mapped to different IP addresses depending on where the DNS requests came from.

The above parameters gives a HyperNet Network Creator some minimum control over the way the network is created. In the example of a video conference HyperNet network where the list of participants is known in advance, the HyperNet Package only needs the Network Creator to provide a participants list and sets the Virtual Network to be not expandable. To deploy a "Youtube-like" video sharing HyperNet network, the HyperNet Package may not ask the Network Creator to provide any configuration information. Instead, The HyperNet Package may be set to expandable and allow anyone to join. In most cases, it is expected that the HyperNet Package comes with a pre-defined configuration file such that the Network Creator needs to do little or no additional editing of the configuration, e.g., this "configuration template" can be in the form of an editable file in the HyperNet Package.

At the same time, our design allows the HyperNet Builders to define their own configuration parameters for their own HyperNet Packages to offer more sophisticated control to the users of their HyperNet Packages. In the following, we list some possible parameters that a Network Creator can "tweak" to manipulate the virtual network. These parameters are specific to the HyperNet Package and will not be interpreted by the Network Hypervisor (i.e., *registerHyperNet()* does not send these parameters to the Network Hypervisor). Note that since these parameters are purely designed and defined by the HyperNet Builder, they can be anything. HyperNet Builders can implement more functionality in their HyperNet Packages and provide more

parameters far beyond the following list for Network Creators to manage their virtual networks.

- *Maximum number of Gateways/Programmable nodes allowed:* This parameter allows the Network Creator to control the scale (as well as cost) of its virtual network.

- *Expand Policy:* Some HyperNet networks might be designed to be expandable in the sense that as new participants join, the HyperNet Package automatically adds new programmable routers to the network to optimize the virtual network topology. Expand policies define rules such as when and where in the network a PR should be added, when should PRs merge, how many participants can share a single gateway, etc.

- *Height of the Tree:* In the case of creating a multicast tree HyperNet network, intuitively, the "higher" the created tree, the more efficient multicast will be. For example, if the height of the tree is 1, the multicast sender will directly connect with each receiver via a transparent tunnel. In this case, "multicast" is the same as "multiple unicast". On the other hand, if we discover all programmable routers in between the sender and each receiver and create a shortest path tree that maps each of the edges onto a physical link between two neighboring programmable routers, the resulting tree will be much "higher" and is much more efficient than the "multiple unicast" tree. This parameter controls the height of the tree.

5.5 HyperNet Participants

We define three types of participants in the HyperNet architecture: (1) *infrastructure participants*, (2) *internet participant* and (3) *third-party participant*.

An *infrastructure participant* is a participant that is in the Virtual Network Infrastructure Provider's network, i.e., is part of the VNIP infrastructure. As a result, the underlying VNIP has some control over this type of participant in the sense that the VNIP can "connect" the participant with any other infrastructure node managed by the VNIP by creating a virtual (or possibly a physical) tunnel between them.

In some cases the VNIP may be able to control/configure the participant node for an infrastructure participant (e.g., set up the routing tables). In other cases, the HyperNet Package may still need to load code onto the infrastructure participant to do the configuration.

An *internet participant* is a standard Internet end system like a home PC. It has conventional Internet access provided by its local ISP and thus can talk to the Network Hypervisor. Compared with infrastructure participants the only difference is that the underlying VNIPs (and thus the Network Hypervisor) do not have control over these participants to automatically map them into the network and configure their routing tables. Consequently, the HyperNet Package will need to load programs on the internet participant and the internet participant needs to voluntarily run those programs in order to "connect" itself to a HyperNet virtual network (i.e., create a virtual tunnel between the participant and its assigned attachment point).

A *third-party participant* is also a standard Internet node except that it does not join on its own. Examples of a third-party participant include existing Internet web sites, DNS servers, and other conventional Internet servers. HyperNet Packages might want to "connect" these types of participants into their HyperNet network without them knowing about it. Consider the case where one wants to redirect Youtube's traffic through one's HyperNet network so that caching and/or compression can be done in the HyperNet network to achieve better performance.

5.5.1 Joining a HyperNet

Different kinds of participants use different mechanisms to join a HyperNet virtual network. More precisely, infrastructure participants and internet participants use "Voluntary Join" to join a HyperNet network while third-party participants use "Involuntary Join" to join a HyperNet network.

5.5.1.1 Voluntary Join

Infrastructure participants and internet participants are typically individuals behind a PC who are willing to download a HyperNet End System package and proactively execute that package to join a HyperNet network. For these two types of participants, joining a HyperNet virtual network means creating a virtual tunnel between the participant and a programmable router that is already in the HyperNet network, and assigning a HyperNet address to the participant so that the participant can communicate with other participants in the HyperNet network using the HyperNet-specific addresses. To illustrate the joining process, consider the example shown in Figure 5.5. In this example, an internet participant B wants to join a HyperNet network created by Network Creator A. Initially (not shown in the figure), A generates a list of participantIDs, and gives the Network Hypervisor this list (via the *registerHyperNet()* API call). A then informs B via some method not specified in the HyperNet Architecture (e.g., via an email or a phone call) about the new virtual network (e.g., the HyperNetName) and the participantID generated for B so that only B knows about its participantID (step 0 in the figure).

Upon receiving the message, B configures the HyperNet configuration file for the corresponding HyperNet End System package to send a join request to the Network Hypervisor. Just like the configuration file for a Network Creator to use to create a HyperNet network, the configuration file for a HyperNet End System package uses a similar format (which is also designed by the HyperNet Builder) so that the

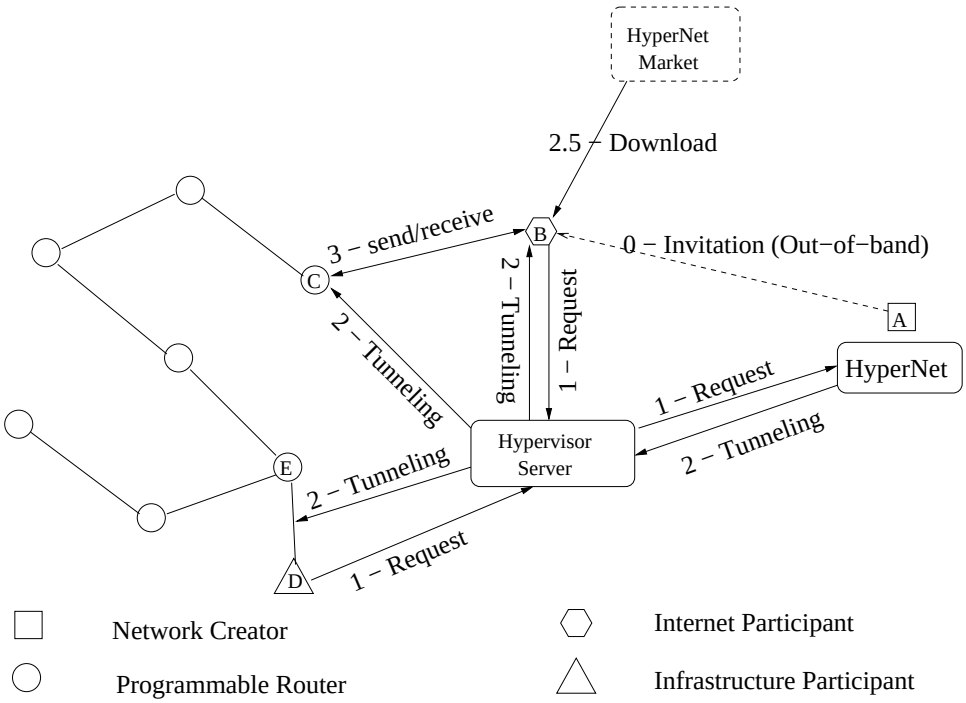


Figure 5.5: Internet Participant and Infrastructure Participant Joining a HyperNet Network

participant can specify, e.g., the name of the HyperNet network he wants to join, his IP address, his credential, and the hypervisor server he wants to use. The join call takes the following form:

TunnelInfo join(HyperNetName, Credential, myInfo)

The join request includes the $HyperNetName$, which uniquely identifies a HyperNet network, B's credential which shows that B has paid to use the HyperNet network, and B's identity information (in $myInfo$) that helps the Network Hypervisor verify that B is authorized by the Network Creator (by checking B's participantID), assign B to an attachment point C, and set up a tunnel between B and C (by making use of B's public IP address).

Upon receiving a join request, the Hypervisor Server checks the validity of B's credential as well as B's participantID in the join request. If the included credential and participantID are valid, the Hypervisor Server then forwards the join request to

the HyperNet Package used by A. After receiving the join request, the HyperNet Package first finds an attachment point C for participant B via the *findPR()* call. It then generates a HyperNet-specific address for B to use to communicate within the virtual network. Finally it sends a "tunneling" message to B containing B's assigned attachment point PR and B's assigned HyperNet-specific address so that B knows who to tunnel through to communicate with the virtual network. The "tunneling" message may optionally include additional end system appliances that B needs to install in order to use the specific virtual network. The HyperNet Package sends the same message (except the appliance information) to C so that C can work with B to successfully set up a tunnel. All communication between the HyperNet Package and Participant B in the joining process goes through the Network Hypervisor.

Any HyperNet Packet sent to B's HyperNet-specific address will be forwarded to B via C. After receiving the "routing" message from the HyperNet Package, B will also set up a local "HyperNet Routing Table" so that any packet destined to any HyperNet-specific address will be forwarded to B's HyperNet attachment point C.

When an infrastructure participant D also wants to join this virtual network, it needs to go through the same process as B, except that when the Network Hypervisor receives the "routing" message from the HyperNet Package, it does not forward it to D and its assigned gateway router. Instead, the Hypervisor Server directly sets up a tunnel between D and the assigned gateway. The Hypervisor Server can achieve this by, for example, assigning the switch interface so that the infrastructure participant is connected to the same VLAN as the assigned gateway.

5.5.1.2 Involuntary Join

Infrastructure participants and Internet participants can voluntarily join themselves to a HyperNet network by invoking the "join()" API call. However, infrastructure participants and Internet participants can also join third-party sites (third-party participants) to the HyperNet network via an "involuntary join". Involuntary joins

are useful when a voluntary participant (an infrastructure participant or an Internet participant) wants to use the HyperNet network to reach a third-party participant that would otherwise never join the HyperNet network. For example, a voluntary participant that wants to access a commercial web site like Google, Facebook, or YouTube is unlikely to get these web sites to join their private HyperNet network. In such cases, the participant can invoke the Network Hypervisor's involuntary join call to "join" these involuntary sites (third-party participants) to their virtual network. The involuntary join call finds the *jumping off point* on the virtual network that is closest to the third-party participant, and then configures the jumping off point to act as a proxy/NAT box forwarding packets to (and from) the third-party participant. The third-party participant is unaware that it is using the HyperNet Package to communicate with the voluntary participant. Finally, the involuntary join API call configures the voluntary participant to route traffic destined for the third-party participant via the jumping off point using the HyperNet-specific address of the jumping off point. This can be achieved by, for example, configuring the local DNS cache of the voluntary participant so that the third-party participant's domain name maps to the jumping off point's HyperNet-specific address.

An Involuntary join request is issued via the following API call:

```
NatInfo joinOther(HyperNetID, Credential, myInfo, unknowingParticipant)
```

Parameter *unknowingParticipant* (a DNS name or IP address) identifies the third-party participant which is unaware of its joining. Upon receiving a *joinOther()* request, the hypervisor checks the credential and forwards the request to the HyperNet Package. The HyperNet Package is in charge of finding a gateway router as the jumping off point for the third-party participant, assigning it a HyperNet address, and setting up routing table entries on all PRs between the requesting participant and the jumping off point PR so that a packet sent from the requesting participant with the third-party's HyperNet address as the destination address will be forwarded

to the jumping off point. In addition, it must set up proper NAT (Network Address Translation) table entries on the jumping off point, and notify the requesting participant (either an infrastructure participant or an internet participant) about the jumping off point. Typically, the HyperNet package will choose the programmable router that is closest to the third-party participant among the PRs that are already in the HyperNet virtual network as the jumping off point (see Figure 5.6). After

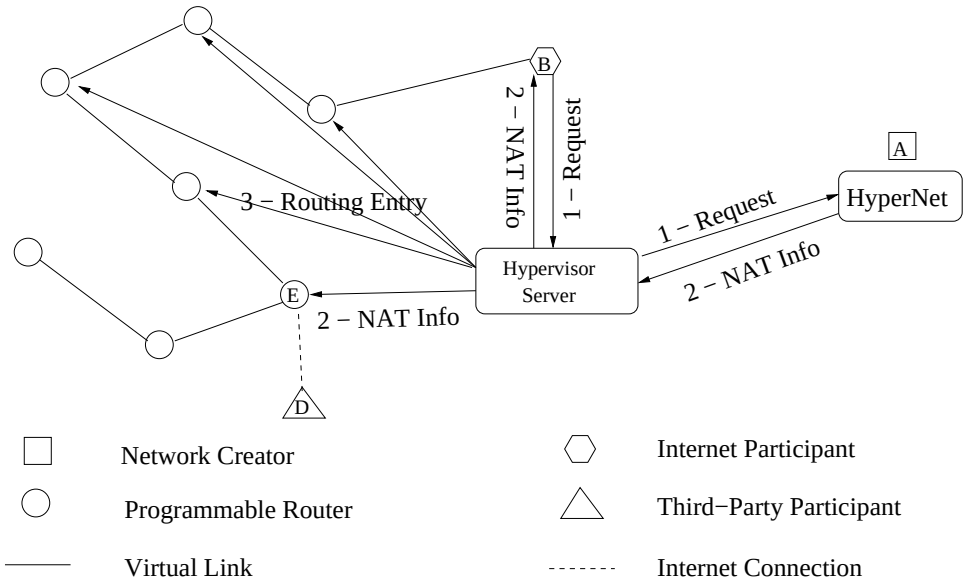


Figure 5.6: Third-Party Participant Joining a HyperNet Network

the NAT box tables are setup on the jumping off point and the routing tables are updated on all PRs between B and E, the selected jumping off point E will intercept all packets that are destined to D's HyperNet address, then replace the destination address with D's IP address, and the source address as E's IP address. It will then send out the packet as a regular IP packet. The third-party participant D replies to E with a regular IP packet, which again gets translated on E to a HyperNet Packet with D's HyperNet address as the source address and the requesting participant's HyperNet address (B's HyperNet address) as the destination address. Third-party participant join only works for the requesting participant (i.e., only B in this case is aware of the HyperNet address assigned to D). Other participants need to call

joinOther() if they also want to communicate with D through the HyperNet virtual network. The HyperNet package first checks whether a HyperNet-specific address and jumping off point PR are already assigned to D and if so, only the routing tables on all PRs between the requesting participant and the previously assigned jumping off point PR need to be updated.

5.5.2 Participant Usage Models

Ultimately a participant must communicate over the HyperNet network it joined. For infrastructure participants, using the HyperNet network for communication is straightforward because the participant's OS is fully integrated into the virtual network (i.e., it uses HyperNet addresses and links). However, internet participants are not fully integrated so that applications need to explicitly choose to use the HyperNet (as opposed to the native Internet) interfaces. We envision four possible ways to implement the end system application code on internet participant nodes.

5.5.2.1 Model 1: Specialized End System Applications

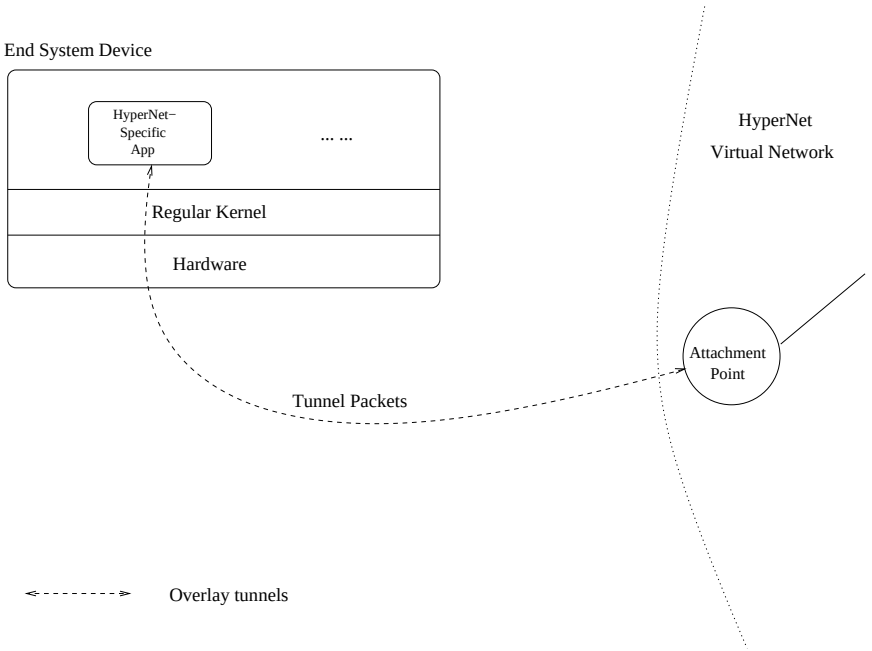


Figure 5.7: Model 1: Specialized End System Application

As indicated by its name, in this model, the HyperNet end system applications are specifically designed and built for one type of HyperNet network. The applications that run on the end system have been built to use the HyperNet's transparent tunnel to tunnel into the HyperNet network. It uses HyperNet address, speaks HyperNet protocols, etc. All the applications needed by the user must be included in the end system part of the HyperNet Package.

As depicted in Figure 5.7, these applications run directly on a conventional kernel. However, the communication between the special application and the attachment point is via an application-specific IP overlay that the application must be fully aware of. Thus, the application needs to be designed so that it supports and knows how to talk the overlay protocol used to reach the attachment point, which also needs to know the overlay protocol.

The major advantage of this usage model is that applications can be designed to run in a standard OS context modulo the need to use the IP overlay for communication. The downside is that such an application is specifically designed only for one type of virtual network. Another drawback of this approach is the overhead of writing HyperNet-specific applications. Ideally, we would like to make use of existing conventional IP applications communicating over the HyperNet virtual network. The next three models explore possible ways to leverage conventional applications.

5.5.2.2 Model 2: Virtual Application Gateways

The idea is to create an *application gateway* that speaks normal IP on one side and HyperNet protocols on the other. The HyperNet side connects via a tunnel to the attachment point. One way to implement this is by creating a virtual interface on the conventional operating system through which a conventional end system application communicates with the HyperNet network. One possibility is to use a Tun/Tap interface [83]. Tun/Tap (Figure 5.8) provides a very convenient interface for intercepting and tweaking IP packets from user space. Conventional

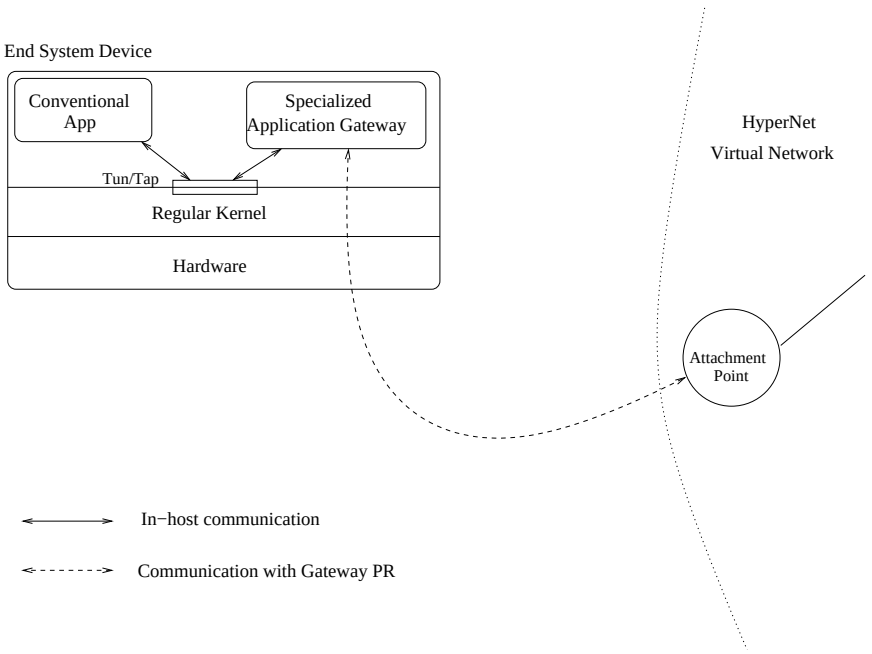


Figure 5.8: Model 2: Tun/Tap Interface

applications send packets to destinations reached through the Tun/Tap interface. The Tun/Tap interface intercepts the packet and sends the packet to a user-space program (the "Specialized Application Gateway" in Figure 5.8). The user-space program then changes the packet format into a format acceptable for transmission across the HyperNet network. One big advantage of the Tun/Tap interface is that it allows end system applications to modify packet headers from user space. Thus, the user space application gateway can modify the packet header so that it uses a totally different network layer header specifically designed to be used and understood by the specialized virtual network (e.g., a HyperNet network that uses non-IP protocols).

In this model, conventional applications can be used, but the HyperNet Builder needs to provide a specialized application gateway in Figure 5.8 to modify packets appropriately for transmission across the HyperNet network. Since the protocols (e.g., packet format) that a HyperNet network uses are network-specific, the functionality of the application gateway is also network-specific and thus, can only be designed by the HyperNet Builder.

5.5.2.3 Model 3: IP-in-IP Tunnels

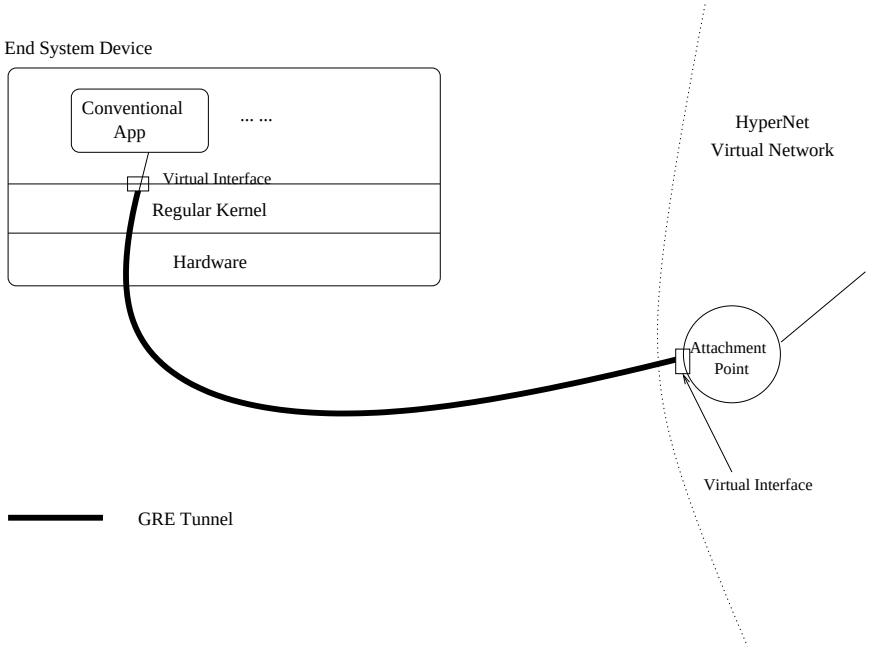


Figure 5.9: Model 3: IP-in-IP Tunnel

In the IP-in-IP tunnel model, the connection between the end system and the attachment point is set up using a GRE tunnel (see Figure 5.9). This model assumes the HyperNet network uses IP and the GRE tunnel extends the HyperNet network to the end system. Conventional applications do not need to be changed, but they do need to use HyperNet IP addresses to talk to PRs in the HyperNet network.

5.5.2.4 Model 4: Virtual Appliances

A virtual appliance is an all-in-one package containing a completely-configured kernel and pre-built software stack/applications; see Figure 5.10. This means instead of modifying or configuring the host kernel, which might require root access (e.g., creating a Tun/Tap interface requires root access), the HyperNet Builder can collect all the modifications or configurations necessary and package them into a virtual appliance.

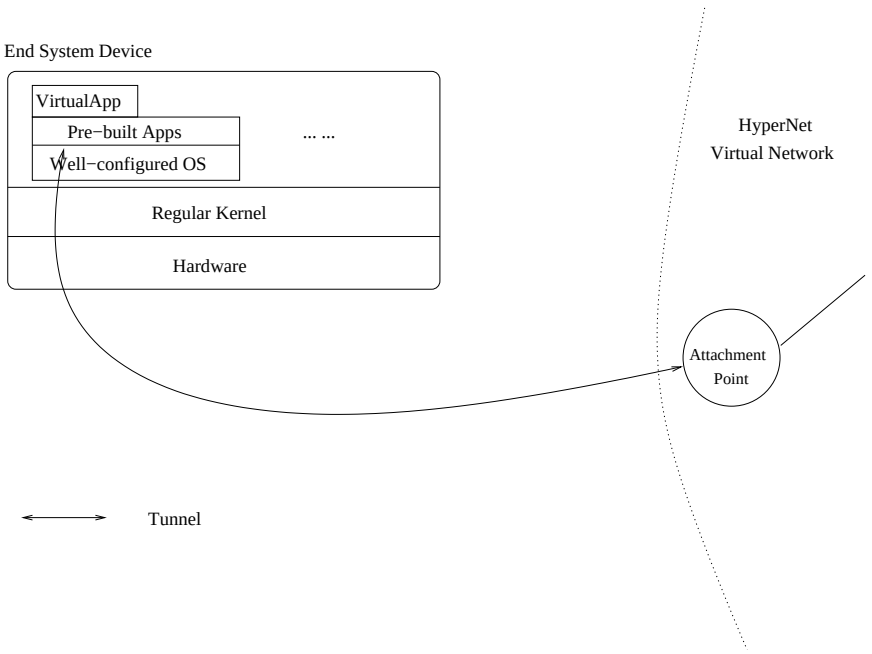


Figure 5.10: Model 4: Virtual Appliance

In this case, the end system part of the HyperNet Package contains a virtual appliance that the user can run in a VM to immediately begin using the HyperNet network. The biggest drawback of this approach is, since a virtual appliance typically includes an entire kernel (sometimes a trimmed kernel), the size of a virtual appliance can be quite big (hundreds of megabytes). On the other hand, it can be run on any end system because it runs in a VM. Indeed, one can imagine many HyperNet Packages using this approach.

5.6 HyperNet DNS Systems

The Internet uses DNS (Domain Name System) to translate human readable names to/from network formatted IP addresses. Similarly, the HyperNet virtual network may also need a DNS implementation to support human readable names. There are basically two ways to achieve domain name translation in our HyperNet architecture: (1) a single DNS used by all HyperNet Packages (a One-Size-Fits-All DNS) and (2) a single DNS for each HyperNet network (a One-For-Each DNS).

The advantages of a One-Size-Fits-All DNS is that each HyperNet Package does not need to include the code to create and maintain a DNS. Instead, they can simply make use of a small number of hypervisor API calls to register and manage their "domain names". The disadvantage of this approach is that the Network Hypervisor needs to create and maintain a HyperNet DNS that can be used by all HyperNet Packages. While caching and a hierarchical structure (just like the current Internet DNS) might help solve the scalability issues, we need to carefully design the system in order to support heterogeneous HyperNet virtual networks with potentially totally different naming and addressing mechanisms.

For the One-For-Each DNS approach, the main advantage is flexibility. First of all, for most HyperNet virtual networks with a small number of participants, a DNS system might not be necessary. The Internet DNS was motivated by the creation of too many machines connected to the Internet. If there are only two participants in the Virtual network, a separate DNS seems to be overkill. Secondly, a local HyperNet DNS name-to-IP file (e.g., /etc/hosts in Linux) is a lot easier and simpler to manage, and a lot faster than a fully loaded DNS. For virtual networks with few participants, an editable DNS name-to-IP file could be sufficient to do name translation. Thirdly, HyperNet virtual networks might use other alternatives to achieve name-to-address translation. The Internet DNS facilitates general-purpose IP communication. For special-purpose virtual networks, other alternatives can be used. For example, in a multi-party video conferencing virtual network, the HyperNet Package could include a "buddy list" so that the participants can name their own "buddies" (i.e., other participants). The main disadvantage of the One-For-Each DNS approach is that for large HyperNet virtual networks requiring a fully functional DNS, the HyperNet Package needs to handle the creation and maintenance of this separate DNS.

5.7 Scalability of the Network Hypervisor

The design of the Network Hypervisor aims to serve a large number of HyperNet Packages at the same time (e.g., the Network Hypervisor might receive HyperNet network creations/tear-downs/updates at the same time, and it also needs to always maintain the runtime status of all active HyperNet networks at any moment). The Network Hypervisor maintains a table containing all necessary pieces of information about each active HyperNet network, from the meta data such as HyperNet names, credentials, etc., to participant-to-gateway mappings. We will describe more details in Chapter 6. It is unlikely that any aggregation of such information will happen among HyperNet networks since all HyperNet networks are different from each other, using different resources, with different participants. Thus, the space (either in memory or on disk) needed in the Network Hypervisor for all HyperNet networks will grow linearly with the number of HyperNet networks running.

The participant of our HyperNet architecture can be any Internet user from anywhere, joining or leaving any HyperNet virtual networks. In regards to scalability one obvious question is, how many participants can join/leave/participate at the same time in our system? The current Internet achieves scalability via DHCP and Local Area Networks – so that the joining and leaving of an Internet user is usually handled by a local DHCP server and the impact is kept within its local area network. While it is obvious that the joining or leaving of a participant only has impact on the targeting HyperNet virtual network it is joining/leaving, our Network Hypervisor also delegates all the handling of a joining/leaving participant to the HyperNet Package. The hypervisor only does a light-weight credential check for each joining participant (or better yet, the hypervisor simply forwards the join request to the HyperNet Package and let the HyperNet Package do credential check), then it forwards the joining request to the HyperNet Package via an upcall. The HyperNet Package then decides how to deal with this new participant – either assigning a new attachment

point for the participant or using an existing programmable router as the attachment point. It may or may not update the topology. After a HyperNet network is deployed, the updating and maintenance task all relies on the HyperNet Package. Thus, the scalability of the Network Hypervisor is not so much the issue as the scalability of each individual HyperNet Package. The hypervisor simply accepts API calls and carries them out. Caching can always be used to help carry out expensive API calls such as finding the shortest path and finding the best attachment point PR. The hypervisor does need to keep track of the resources consumed by each HyperNet network for billing purposes and as a result, the (memory/disk) space cost grows as each HyperNet network grows. In theory, the hypervisor could delegate this task to the underlying VNIPs to spread the work load.

Finally, what is the scalability of each of the hypervisor API calls? For example, how much time will it take to find a nearby programmable router or to find an optimum path between two PRs? As mentioned earlier, the hypervisor can distribute the task of finding nearby PRs to VNIPs by maintaining a IP address to VNIP mapping (the hypervisor can get this mapping from e.g., the current DNS system). The size of this mapping increases with the number of VNIPs and the number of IP address segments. To find the shortest path between two programmable routers, one can easily use a simple implementation of Dijkstra's algorithm that has a complexity of $O(N^2)$, with N being the number of nodes in the graph. To find K shortest paths between two nodes, one can run Dijkstra's algorithm K times with an approximate complexity of $O(KN^2)$. While it is unrealistic to compute paths on demand, one can pre-compute the paths and cache them with the assumption that the physical topology will not change very often. Moreover, the hypervisor can map the task of finding a PR-to-PR path to concatenating intra-VNIP paths with inter-VNIP paths to further decrease the amount of time needed (with intra-VNIP and inter-VNIP paths pre-computed).

Chapter 6

A Prototype Implementation

To demonstrate the HyperNet architecture, we have implemented a Network Hypervisor prototype using GENI [23] as the VNIP. As mentioned in Section 2.2, GENI provides an Internet-scale network testbed for researchers to create experimental networks that are isolated from each other. Multiple control frameworks exist in GENI, including PlanetLab [20], ProtoGENI [52] (previously known as Emulab [21]), InstaGENI [84] (an extension of ProtoGENI), ExoGENI (an extension of ORCA [53]) and ORBIT [54]. Each control framework has multiple instantiations called *aggregates*. For example, ProtoGENI and InstaGENI have a Kentucky Aggregate, a Utah Aggregate, and several others. ExoGENI has the UNC RENCI Aggregate and the BBN Aggregate. Each aggregate has its own aggregate manager to manage its resources. In HyperNet terms, each aggregate can be thought of as a separate VNIP. To create a virtual network across multiple aggregates in GENI, an expert user needs to communicate with all the corresponding aggregate managers to reserve and connect resources to form a topology. A virtual network instance created in GENI is called a "slice", and each programmable node in a slice is called a "sliver". A sliver may be in the form of a virtual machine or a physical machine, and can act as either a PC or a router. While keeping their own proprietary APIs to control and manage the resources, most of the GENI control frameworks have also implemented the GENI AM API [76], which provides function calls to: (1) discover available resources, (2) query

capacity information about resources, (3) create/delete/update/renew a sliver, (4) register/unregister/shutdown/renew a slice, and (5) check the status of a sliver/slice.

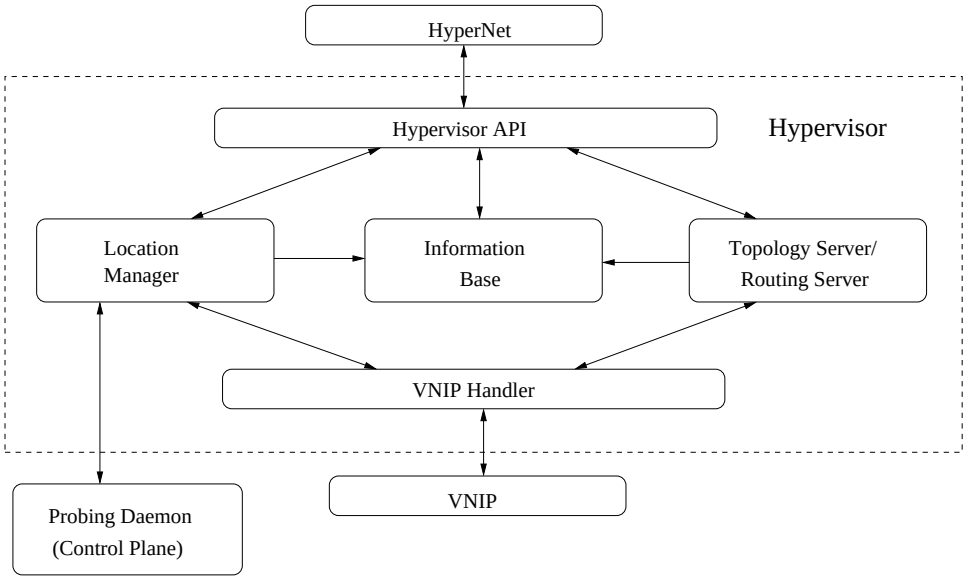


Figure 6.1: Hypervisor Implementation

Our prototype Network Hypervisor uses the GENI aggregates as VNIPs, making AM API calls to the aggregates to reserve resources and create topologies. As shown in Figure 6.1, our Network Hypervisor implementation includes a VNIP handler at the bottom layer. The VNIP handler talks to different VNIPs (via the VNIP API provided by each VNIPs – in our case, the GENI AM API[1]) to discover, manage and monitor VNIP (GENI Aggregate) resources. The top layer of the hypervisor implementation is the set of Hypervisor API calls described earlier. The Hypervisor API helps Network Creators create specialized virtual networks. It also helps the participants join and use virtual networks. To make use of the VNIP information and support the Hypervisor API calls, the hypervisor implementation includes three important components: an *Information Base*, a *Location Manager* and a *Topology Server/Routing Server(TS/RS)*.

[1]An earlier implementation of the Network Hypervisor used the ProtoGENI API to talk with ProtoGENI aggregates — just to prove that our implementation is able to support multiple VNIP protocols.

6.1 The Information Base

The Information Base component maintains information about VNIPs needed to implement various Network Hypervisor API calls. It contains the location information about each of the underlying VNIPs and the most up-to-date topology information of the VNIPs. In addition, for each active HyperNet instance (i.e., currently executing HyperNet Package), the information base maintains a table which includes: the HyperNet name, the participant ID list, creator information, active participants information (e.g., their IDs, IP addresses, HyperNet addresses, and their corresponding attachment points), the virtual network topology (such as reserved nodes and links), a credential (which is verified whenever a new participant that does not have a valid participant ID joins), and other monitoring information such as total active time, resources consumed, and total cost. A NAT table is also maintained for each of the running HyperNet networks containing the HyperNet name, third-party participants (the IP address of the third-party participant), the assigned jumping off point PRs, and the third-party participants' assigned HyperNet-specific addresses.

6.2 The Location Manager

The Location Manager determines the the closest attachment point to a participant. The location manager fetches location information about each VNIP through the VNIP handler and saves it in the Information Base when the Network Hypervisor starts up. A VNIP's location information includes the VNIP's local view of nearby end hosts (e.g., IP address prefixes). Just as today's ISPs configure local DNS servers for their customers and thus "know" their nearby customers (or at least know that the local DNS server is next to certain customers), the Network Hypervisor can obtain this information from its VNIPs. The location manager also coordinates the task of discovering the network location of a participant among nearby VNIPs. It fetches

the information about how close each VNIP is to the participant (i.e., the network probing result from one of the VNIP infrastructure nodes to the participant). The location manager also caches the results. Finally the location manager can determine the PR with the best network performance (e.g., the smallest Round Trip Time) near a participant and return that programmable node in the findPR() API call. In our implementation, we consider each of the GENI aggregates as a distinct VNIP since different aggregates typically sit in different geographical locations, use different sets of IP addresses, and manage resources only within their own aggregate.

To implement the location manager, we created a long-lived experiment in GENI, in which we reserved one PR from each aggregate to assist the location management service. The reserved PRs are used by the Network Hypervisor as a "control plane" to gather information for the location manager (see Figure 6.1). Since the Network Hypervisor has control over each of the reserved nodes, the Network Hypervisor can load a "probing application" onto those nodes and fetch network performance information between the reserved "probe" nodes and any participant. The probing results from each of those nodes represent the "closeness/nearness" of their corresponding aggregates (VNIPs) to the participant. The location manager then picks an available PR from the closest VNIP which satisfies the capacity requirements set by the HyperNet Package. In this way, the *findPR()* API call discovers the closest PR from a set of VNIPs for each participant.

6.3 The Topology Server/Routing Server (TS/RS)

To support the *seeTopo()*, *getShortestPath()*, *findCentralNode()*, *drillDown()* Network Hypervisor API calls (used by HyperNet Packages to explore the topology of the underlying VNIPs), we designed and implemented the *Topology Server/Routing Server (TS/RS)* component. The TS/RS's job is to obtain the topology information

from the underlying VNIPs[2], save it in the information base, and implement the corresponding hypervisor API calls. Since the *seeTopo()* API call returns a topology with dynamic information about the availability of the resources in the topology (i.e., programmable nodes might be unavailable or links might be congested), the topology server should update the (available) topology information in the information base whenever a new HyperNet virtual network is created or a virtual network is torn down. On the other hand, the underlying VNIP may also sell its resources to customers other than the hypervisor. Thus, the topology server should also be responsible for maintaining an up-to-date topology from the VNIP, either by continuously pulling topology information from the VNIP or having the VNIP pushing updated topology information to the hypervisor upon any changes. The Routing Server makes use of the topology information in the information base and accomplishes tasks such as calculating paths between two nodes, finding a central node among a set of nodes, and building a multicast tree connecting a set of leaf nodes.

6.3.1 Finding a Central Node

The algorithm to find the central node in between a set of nodes is simple. If the provided set of nodes are all in the same aggregate, first of all, for each pair of the nodes in the node set, the hypervisor finds the shortest path between them (assuming all paths are symmetric and all virtual links have the same distance). Next, the most popular node (i.e., the node that appears most frequently among all shortest paths) is chosen as the candidate central node. Finally, the characteristics of the candidate node is checked. If it satisfies the central node requirements (processing power, disk space, etc.), it is returned as the central node, otherwise the second most popular node is chosen as the candidate node. This process is repeated until it finds a candidate node which satisfies the central node requirements. If the given set of nodes are from

[2]VNIP topology information contains the topology that a VNIP exposes to the hypervisor via the VNIP API. It does not have to be the physical topology. In cases where the VNIP wants to hide part of its physical topology, virtual tunnels can be returned by the VNIP.

multiple aggregates, then the TS/RS uses the location manager to figure out the best aggregate that provides the minimum average RTT to all given nodes. Then an available PR is selected from that aggregate to serve as the central node.

6.4 Random Topology Generator

In ProtoGENI (as well as in InstaGENI and ExoGENI), any two programmable nodes in the same aggregate can be directly connected via a VLAN without going through any other nodes. In other words, ProtoGENI nodes in the same aggregate form a fully connected graph – which is not reflective of real-world topologies and uninteresting from an experimental standpoint. Moreover, the connection between two aggregates in ProtoGENI is typically set up via one, or occasionally more, IP tunnels. Thus, the best way to connect two programmable nodes in different aggregates is to directly connect them via an IP tunnel (e.g., ProtoGENI's solution is a GRE tunnel).

In PlanetLab, the same holds true in the sense that the best way to connect two PlanetLab nodes is also to directly connect them without going through any other PlanetLab nodes. The only difference is that this "connection" is accomplished via an overlay channel (e.g., a TCP connection or a UDP connection) instead of a VLAN or a GRE tunnel. In short, nodes in both ProtoGENI (as well as InstaGENI and ExoGENI) and PlanetLab form fully connected topologies and so using them as VNIPs is uninteresting.

To make the topologies more interesting, we intentionally removed some of the links from the topology after the Network Hypervisor fetches the topology from GENI. Removing links from the topology helped us to avoid the uninteresting case where the topology is fully connected. As a result, the observed topology might not be fully connected. In our implementation, we developed a random topology generator on top of ProtoGENI. It randomly generates a topology connecting all available resources within a ProtoGENI aggregate. The generated topology looks like a physical point-

to-point topology, hiding the fact that the resources actually form a fully connected topology. Next, we program the hypervisor API calls used to explore the underlying physical topology to return information from this randomly generated topology rather than the actual fully connected topology.

6.5 Hypervisor Performance

We implemented our Network Hypervisor using Java as the programming language. We used about 5000 lines of Java code to implement all the hypervisor API calls as well as the HyperNet library calls described in early chapters. The source code as well as the HyperNet Builder manual can be found at [85]. To prove that our designed API calls are useful and adequate to compose a variety of HyperNet Packages, we implemented four different types of example HyperNet packages, each can be used to deploy a specialized HyperNet network. We will describe our example HyperNet Packages in Chapter 7. Our next objective is to measure the performance of our hypervisor service. More precisely, we wanted to know: (1) how long it takes for the Network Hypervisor to create the virtual network topology specification that can be given to the underlying VNIP, (2) how long it takes to deploy a virtual network via the Network Hypervisor onto the physical infrastructure provided by the VNIP, and (3) whether the Network Hypervisor can handle concurrent requests. If so, what is the performance? To answer the above questions, we designed several "stress tests" that we applied to the Network Hypervisor.

6.5.1 Experimental Context

We implemented our Network Hypervisor on a virtual machine running Ubuntu 12.04, hosted on an Intel Xeon CPU E5520 running at 2.27 GHz. The virtual machine uses 1GB memory. Because the hypervisor implementation is coded in Java, the HyperNet Package can be run on any platform. Our Network Creator was a physical PC running

MAC OS X, with a 2.4GHZ Intel Core 2 Duo CPU and 8GB DDR3 memory. We tested with both Java Runtime Environment (JRE) version 1.6 and 1.7 to run the server as well as the HyperNet Package. The results are similar. Communication between the Network Creator and the Network Hypervisor is via XML-RPC, with a round-trip time of about 0.5 milliseconds. As stated earlier, We used several GENI aggregates as the VNIPs.

6.5.2 Build Time

Build time measures the interval between the time from when the Network Creator starts executing a HyperNet Package and the time when the Network Creator creates a topology description file describing the virtual network to be instantiated. In GENI's context, the "topology description" file is an "RSpec" file. During this process, the Network Hypervisor handles HyperNet registration request, initializes all the corresponding tables for the HyperNet network (including the participant list, the join request list, the attachment point table and the status table), creates the RSpec file for the virtual network, saves the file to a per-HyperNet directory and returns the RSpec to the caller (the Network Creator). This step does not include any queries to the GENI aggregate manager.

The performance results are shown in Figure 6.2. In this set of experiments, we use the *buildRandomRing()* HyperNet library call to create ring topologies. The X axis shows the number of nodes in the ring topology. The Y axis measures the build time. The RSpec file has length O(n): there are n nodes and n connection links. From the results we can see that the time spent creating a ring topology increases with the number of nodes on the ring. However, to create a ring topology with almost 400 nodes, our Network Hypervisor only used about 8 seconds. Now let us take a look at the time needed for a VNIP to actually allocate, initialize, and deploy the network.

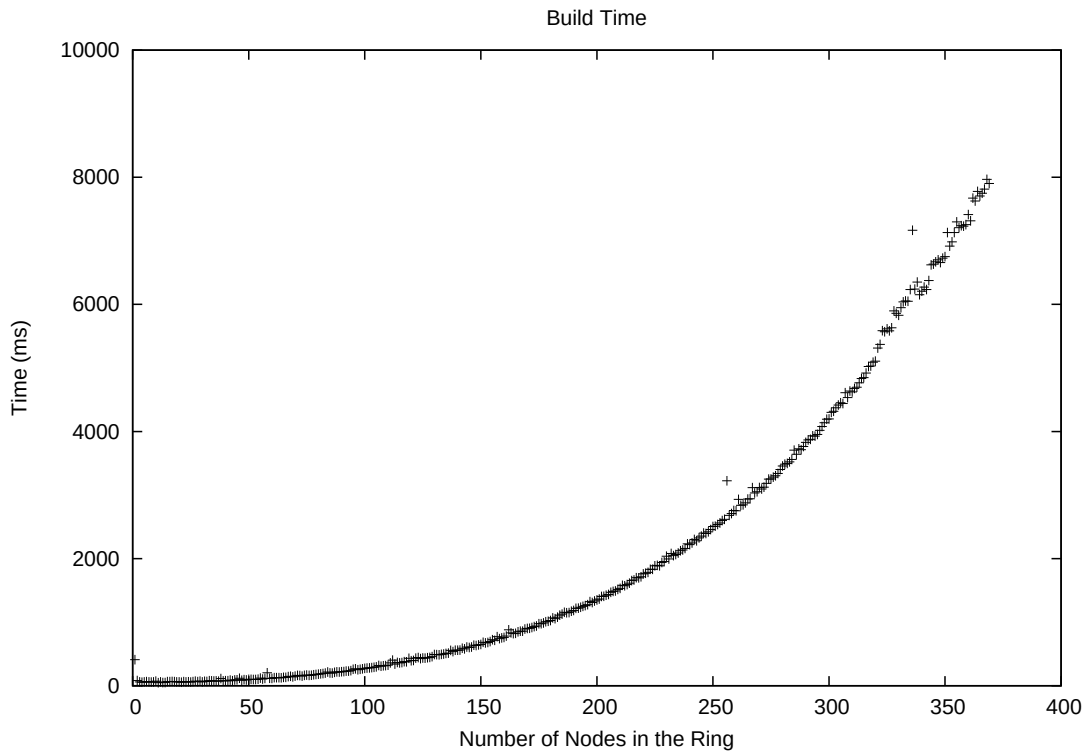


Figure 6.2: Time spent building a HyperNet Ring Topology

6.5.3 HyperNet Deployment Time

HyperNet *deployment time* measures the interval between the time when the Network Hypervisor gives the topology description file to the underlying VNIP and the time when the VNIP has mapped the description onto the infrastructure, the resources are reserved, booted, configured, and ready for use. This step is outside the control of the Network Hypervisor; it is purely the responsibility of the VNIP. It typically includes:

1. Identifying the nodes that need to be included and then reserving them – in the GENI context, the RSpec file can specify particular nodes to include; GENI calls them "bound resources". The RSpec file can also specify that it only needs a node, in which case GENI may choose any node; GENI calls these nodes "unbound resources". We have done experiments using both bound and unbound resources and the results are similar. Two types of nodes are

currently available in GENI: physical PCs (a category that includes physical PCs, physical routers, and physical OpenFlow Switches) and Virtual Machines (VMs) (a category that includes virtual routers and virtual nodes created by OpenVZ, KVM or Xen).

2. Identifying and reserving the requested virtual links that connect the reserved nodes. The VNIP typically first finds a physical path between the two nodes, then it uses its own control plane to configure all switches and/or routers along the physical path to build a virtual link. Each VNIP has its own techniques. GENI creates a virtual link within an aggregate using VLANs, and it creates a virtual link between two nodes in different aggregates using GRE tunnels. It can also employ *stitching*, an inter-domain VLAN technique that requires all aggregates along the way to co-operate to create "stitching VLANs". In our experiments, we only use resources from one aggregate – the Kentucky Aggregate. So GENI only needed to deal with VLAN creations within the same aggregate. Deploy time on other aggregates should have similar results. We used the GENI Aggregate Manager API to deploy the virtual network. For virtual network topologies that span across multiple aggregates, we would do the deployment requests simultaneously on multiple aggregates, so we assume that the deploy time should be similar to the case of one aggregate. Stitching VLAN links between aggregates takes GENI longer time because it needs to calculate the aggregate-level path between the two aggregates and then configure each aggregate sequentially (and if one fails, GENI might wait for up to 20 minutes and then try again). Since stitching is still a new (and not stable) GENI technique, we did not do any performance test requiring stitching.

The results of deploy time for ring topologies on a GENI aggregate are shown in Figure 6.3. There is little difference between the time spent in deploying a ring topology with 3 nodes and a ring topology with 7 nodes. GENI's control

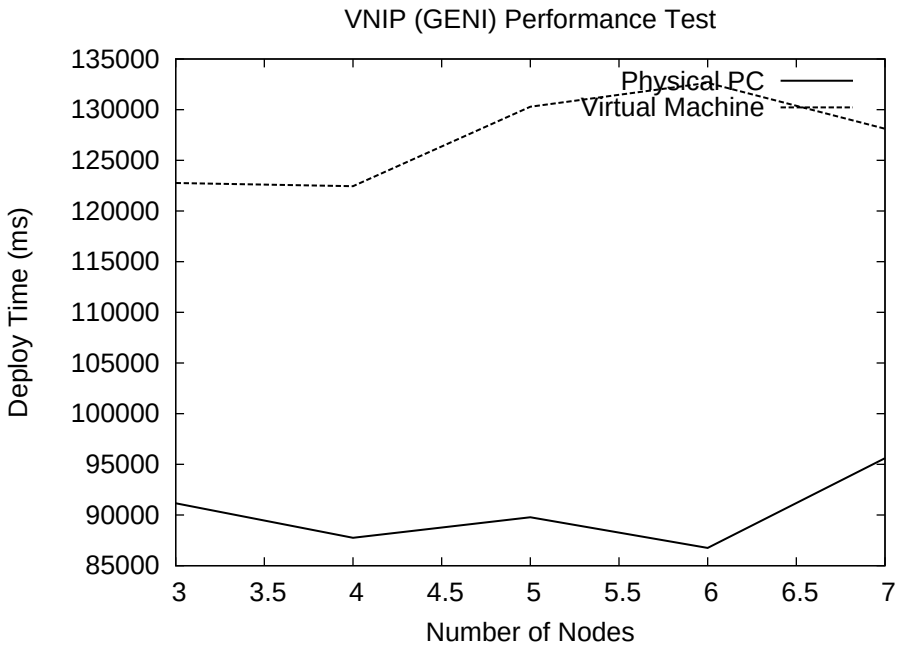


Figure 6.3: Deploy Time for Ring Topologies in a GENI Aggregate

plane (the Aggregate Manager) queries for resources and instructs nodes to boot-up simultaneously. However, there is a significant time increase (over 30 seconds) if the requested resources are virtual machines. This delay is because the aggregate manager needs several extra steps to prepare/configure a virtual machine. For example, it needs to identify the virtualization techniques to be used to create the virtual machine and it also needs to create a profile for each virtual machine to specify the resources (i.e., CPU, Memory, and disk space) to allocate to the virtual machine. More importantly, if we compare the deploy time with the build time in Figure 6.2, we can conclude that the build time is almost nothing compared with the deploy time. Of course, GENI may well improve in the future, and other VNIPs may be more efficient.

6.5.4 Concurrency Test

Multiple HyperNet Packages could call the Network Hypervisor APIs at the same time and thus impose a heavy load on the Network Hypervisor. Although we

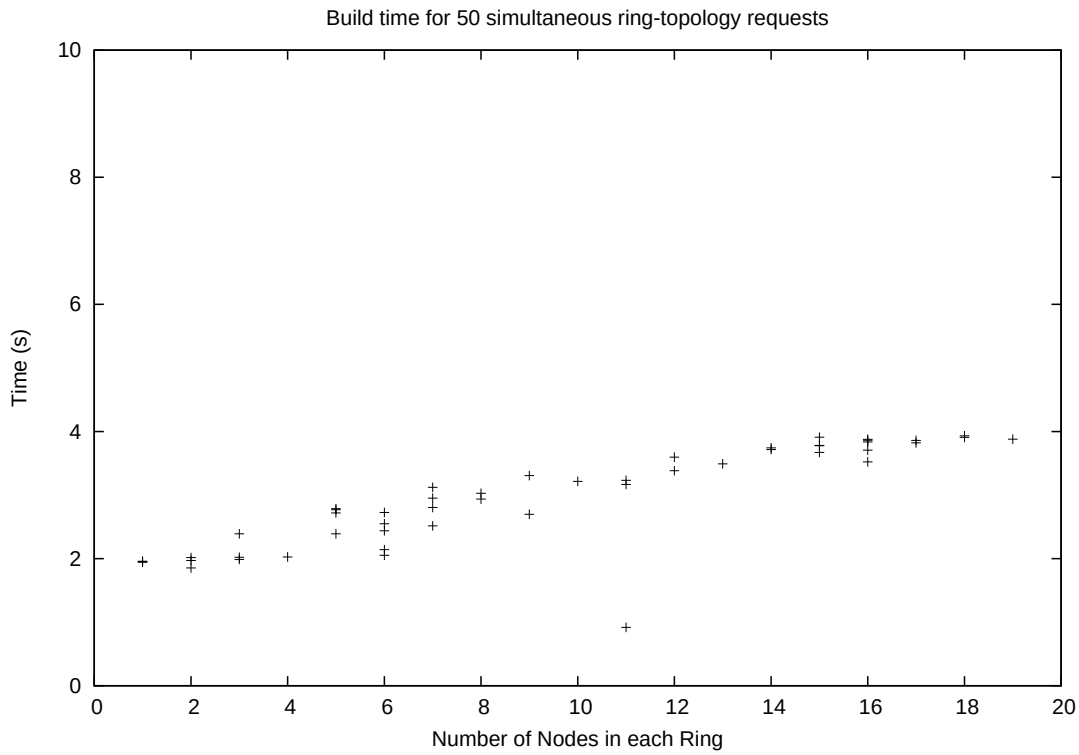


Figure 6.4: Time spent in deploying 50 HyperNet Topologies with concurrent requests

could implement our Network Hypervisor on multiple hosts with queuing and load balancing, we were interested in knowing the performance of a single Network Hypervisor handling concurrent requests.

In this set of experiments, we ran multiple HyperNet packages simultaneously. Each package asked the Network Hypervisor to create a Ring topology with a randomly selected number of nodes (from 1 node to 20 nodes). We recorded the build time for each request.

Figure 6.4 shows the build times when the Network Hypervisor receives 50 concurrent *buildRandomRing()* requests. Figure 6.5 shows the build times when the Network Hypervisor receives 100 concurrent *buildRandomRing()* requests. Figure 6.6 shows the build times when the Network Hypervisor receives 200 concurrent *buildRandomRing()* requests. In each set of experiments, the test script creates the corresponding number of threads and each thread acts as a Network Creator executing

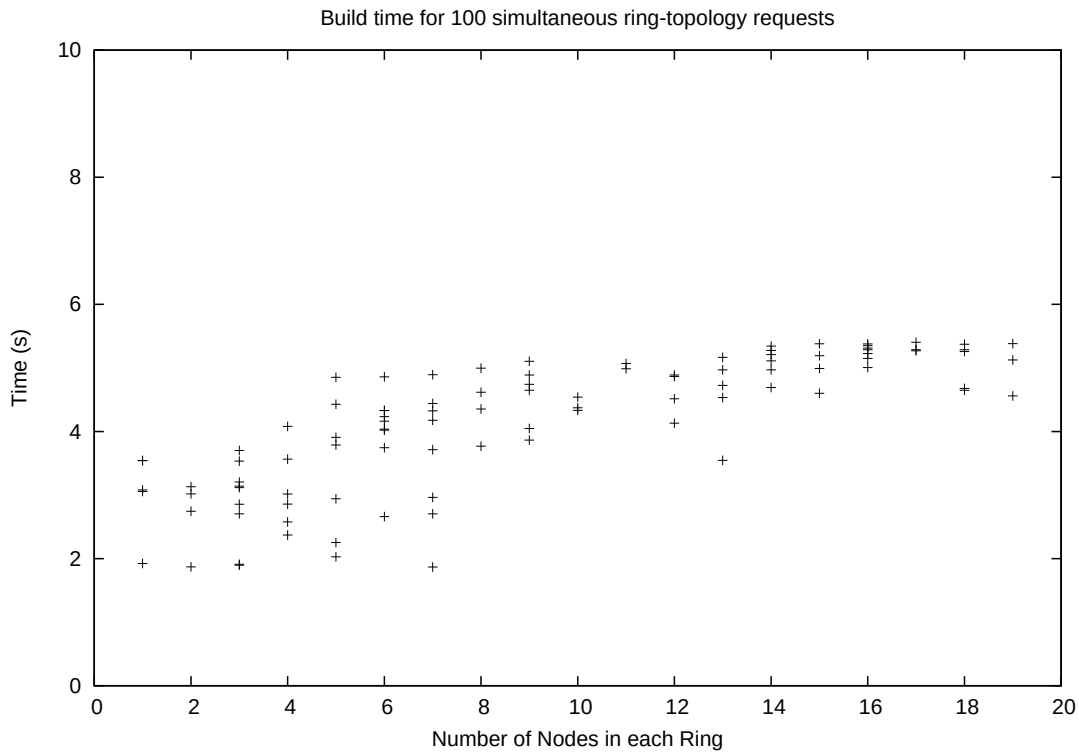


Figure 6.5: Time spent in deploying 100 HyperNet Topologies with concurrent requests

a package that requests to create a ring topology with a given number (picked from 1 to 20) of nodes. The test script needed 159 ms to create 50 threads, 233 ms to create 100 threads, and 658 ms to create 200 threads.

This series of graphs show that the larger the number of simultaneous requests, the longer it takes for the Network Hypervisor to fulfill each request. In general, the larger the number of nodes in the requested topology, the longer the build time. Most importantly, even when there are 200 requests coming into the Network Hypervisor simultaneously, the Hypervisor can still manage to handle all the requests in less than 10 seconds apiece. If we increase the number of concurrent threads on the test script, it is possible that the server will fail with "connection reset" exceptions, because the XML-RPC library we used in our Network Hypervisor supports a limited number of concurrent TCP connections. For comparison, we did the same experiment using sequential requests. In this case, instead of creating 200 threads at the same time, the

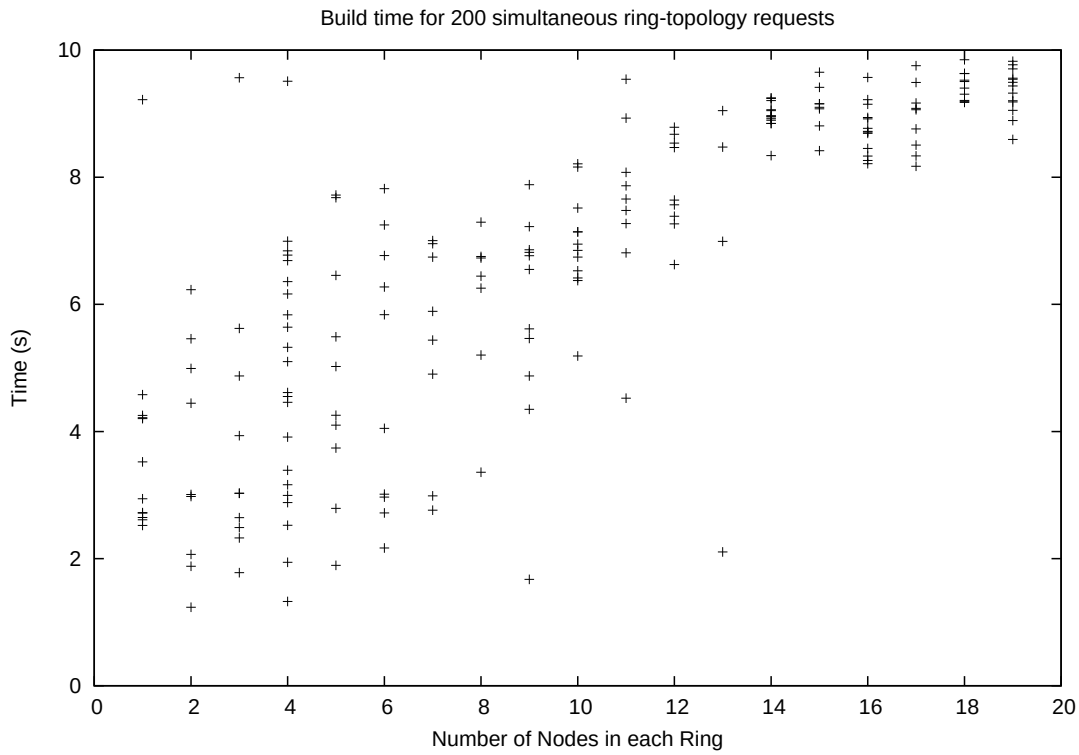


Figure 6.6: Time spent in deploying 200 HyperNet Topologies with concurrent requests

client waits until each thread successfully terminated (a thread will terminate when the Network Hypervisor reports that it has accomplished the build phase) before creating the next thread. The result is shown in Figure 6.7. From the result we can see that the build time for each the request is within 0.5 seconds. The cumulated build time for all 200 sequential requests is about 7.5 seconds (the graph only shows the build time for individual request). This result indicates that sequential request handling out-performs concurrent request handling. The worst case in the sequential request handling is for a request to wait for 7.5 seconds before it gets executed–which still out-performs the 10 seconds worst case for concurrent request handling. As a result, if we want to deploy a production Network Hypervisor, we would want to use queuing technique to sequentially handle concurrent requests.

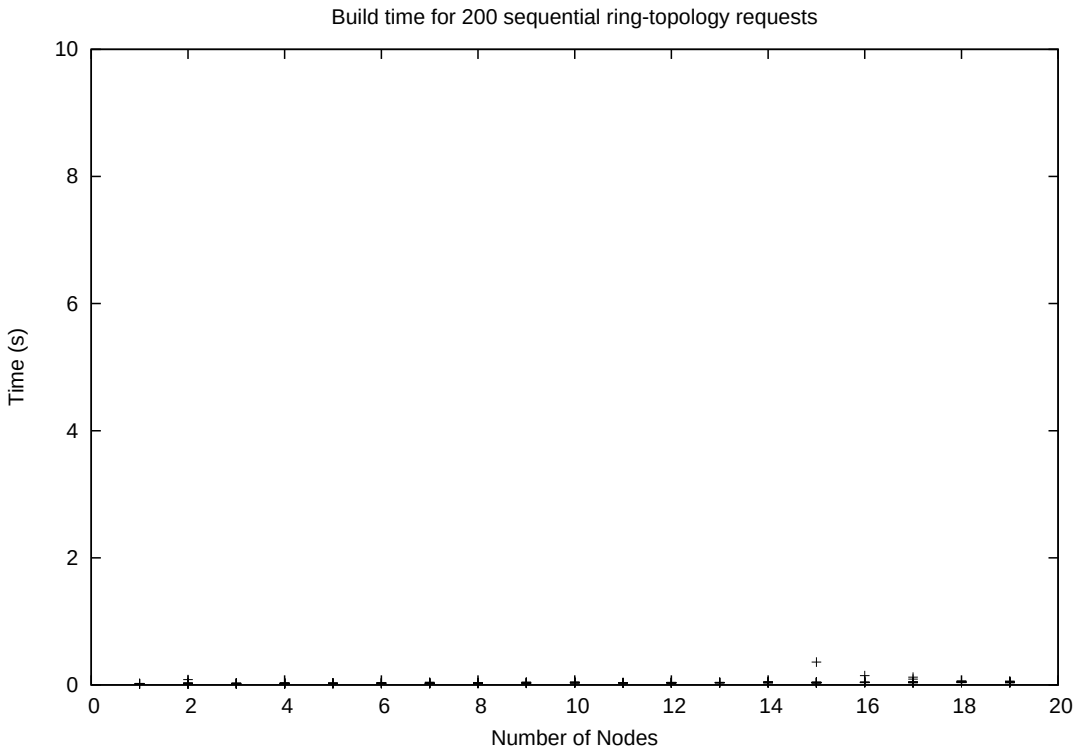


Figure 6.7: Time spent in deploying HyperNet Topologies with sequential requests

Chapter 7

Example HyperNet Packages

To demonstrate the concept of a HyperNet Package, we created several example HyperNet Packages, including: a Multicast HyperNet, a MobileNet HyperNet, a GameNet HyperNet, and an OpenFlow Load Balancing HyperNet. We tested all the packages using the Network Hypervisor implementation described in Chapter 6.

7.1 A Multicast HyperNet

The multicast HyperNet Package dynamically creates an any-to-any multicast network specifically tailored to the participants in the multicast group. It supports native IP multicast applications and because it uses tunnels, it works even over, or between, VNIPs that do not support IP multicast. The HyperNet Package includes both router software to be loaded onto intermediate routers and an end system package which is used by a HyperNet Participant to join and use the multicast virtual network. Generally speaking, the multicast HyperNet Package contains three parts: a network creator-defined HyperNet configuration file, the software stacks necessary to support the multicast protocol (in this case, we used the Protocol Independent Multicast - Sparse Mode (PIM-SM) multicast protocol [30]), and a Java program that makes use of the Network Hypervisor APIs to build and deploy the Multicast network.

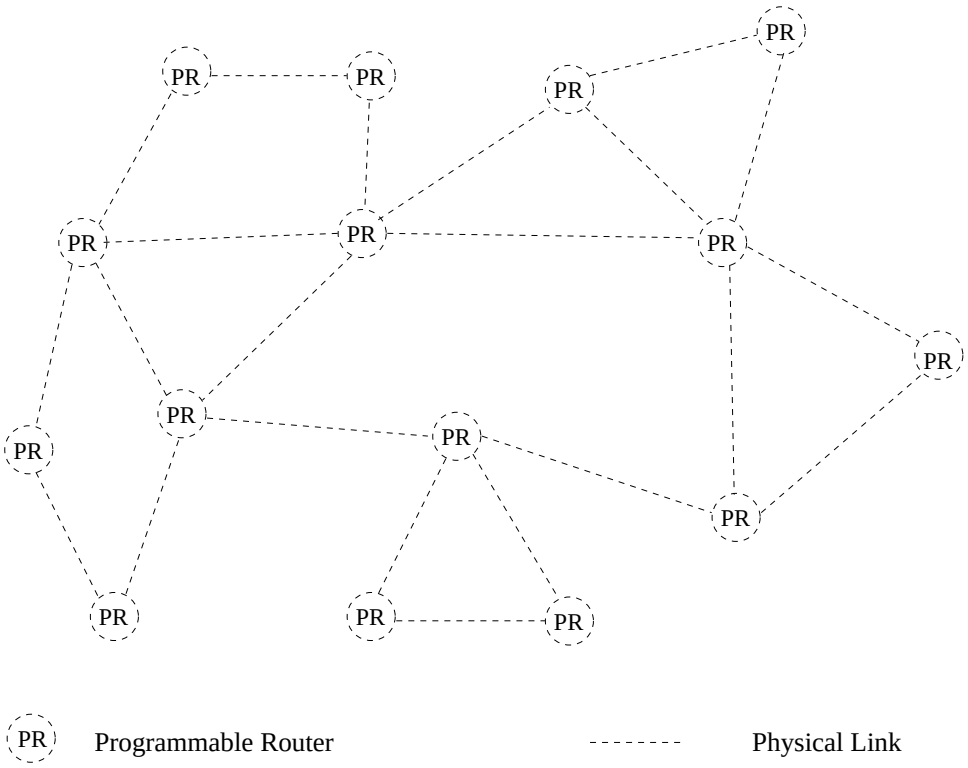


Figure 7.1: (Generated) Physical Topology of the VNIP

Before a Network Creator executes the Multicast HyperNet Package, it needs to first configure the configuration file for the HyperNet Package. The configuration file includes the name of the HyperNet network, the expected participants of the network and their corresponding participant IDs, and the credentials of the HyperNet network which is used by the hypervisor to check the legitimacy of a joiner. The Multicast HyperNet Package first uploads the HyperNet configuration to the Network Hypervisor to register the HyperNet network. It then finds an appropriate attachment point for each expected participant from the configuration file. Next, the HyperNet Package calls the HyperNet Library API *buildRPTree()* to create a tree (reserving all nodes in the tree topology) that connects the attachment point routers as leaves of the multicast tree. Finally, it loads appropriate multicast software stacks and configuration files onto each node of the tree, and requests the Network Hypervisor to deploy the HyperNet network. Figure 7.1 shows our experimental VNIP resource,

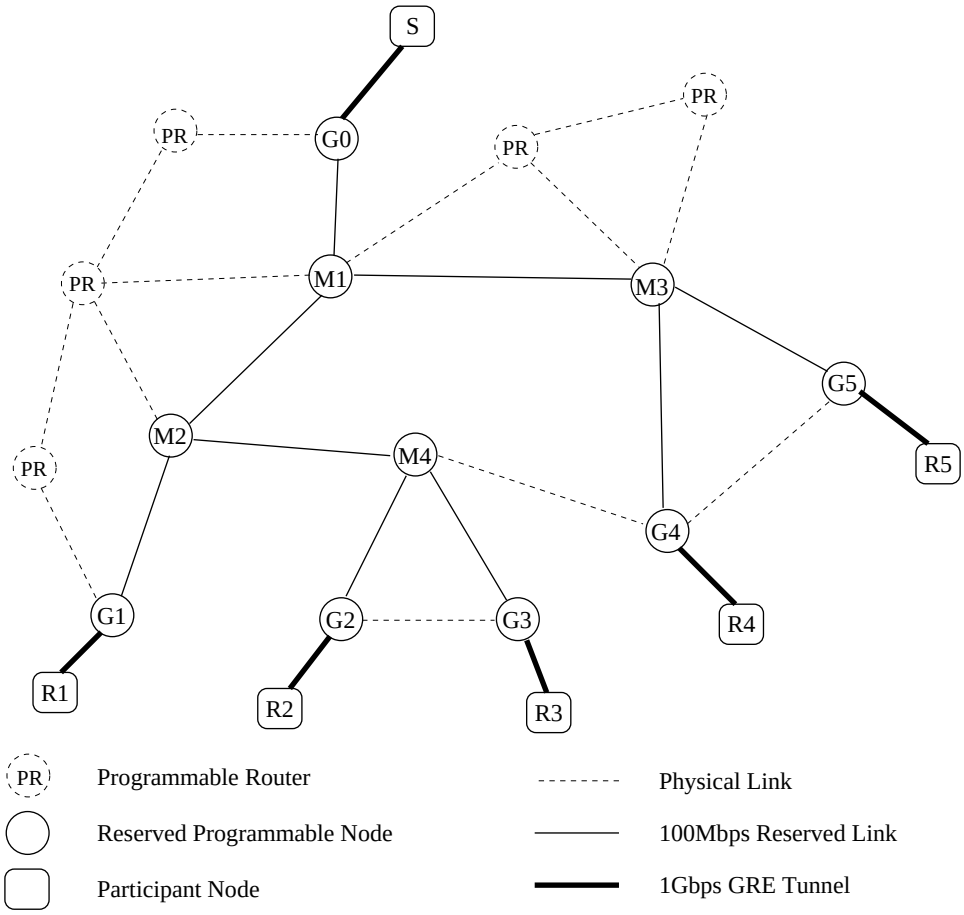


Figure 7.2: Reserved Multicast Topology

and Figure 7.2 shows the reserved nodes and links. G0–G5 are attachment point gateways and M1–M4 are multicast routers connecting the attachment points. S is the sender of the multicast tree and R1–R5 are receivers of in multicast tree (in fact, this multicast tree also allows any receiver to send multicast packets to other nodes). The pseudo code for the HyperNet Package is illustrated in Algorithm 1, with the actual Java code slightly exceeding 200 lines.

Algorithm 1: The Multicast HyperNet Package

```
myConfig = readConfig() //Read configuration file
regHyperNet(myConfig) //Register HyperNet
for every participant p in myConfig do
   gateway = findPR(p)
   gatewayList.add(gateway)
end for
myTree = buildRPTree(gatewayList)
for each programmable router pr in myTree do
   loadApp(pr, PIMSMApp) //load PIM Sparse Mode Application onto each pr
end for
buildTopo() //deploy the topology
//Join Request Handling Process
while true do
   joinRequest = checkJoin() //get join request
   tunnelInfo = createTunnelInfo(joinRequest)
   //set up tunnel between gateway and new participant
   addGW(tunnelInfo)
end while
```

Algorithm 2: The Multicast HyperNet End System Package

```
myConfig = readConfig() //Read End System configuration file
tunnelInfo = join(myConfig) //join request
configInterface(tunnelInfo) //use the returned tunnel
        //information to set up tunnel with
        //the corresponding gateway
```

The sender and receivers are all participants in the HyperNet network. They connect to their corresponding gateway routers via the Multicast HyperNet End System package. Just like the HyperNet Package, the End System package also includes a configuration file, in which a participant specifies its participant ID. As

shown in Figure 7.2, a legitimate participant successfully joins the multicast HyperNet network by creating a GRE tunnel from the end system to the corresponding attachment point.

Our experiment uses the *pimd* multicast daemon [82] and we ran PIM-SM multicast on each programmable node, with $M1$ as the Rendezvous Point selected by the HyperNet Package code. The pseudo code for the HyperNet End System package is illustrated in Algorithm 2, with the real code slightly exceeding 50 lines.

7.1.1 Multicast Results

Table 7.1 shows the amount of time consumed in each step of building the multicast topology. We created the same multicast tree 8 times. On average, the time consumed

Table 7.1: Time Required to create a Multicast Virtual Network

	Build Time		Deploy Time	
Exp. #	Find Gateway (s)	Build Tree (s)	Deploy Network (s)	Join (s)
1	2.26	0.518	100	0.685
2	2.26	0.44	101	0.679
3	2.28	0.458	103	0.699
4	2.28	0.416	102	0.673
5	2.28	0.425	102	0.659
6	2.26	0.43	81.6	0.614
7	2.26	0.425	82.5	0.696
8	2.3	0.438	104	0.831
StdDev.	0.015	0.033	9.25	0.06
Avg.	2.27	0.444	97	0.692

to find a gateway router for each expected participant is just over 2 seconds. In our implementation, in order to choose the nearest attachment point for a participant, the location manager in the hypervisor made use of two representative nodes in the Utah aggregate and the UK aggregate to get the average round trip time from each representative to the participant using 50 ping probes when the HyperNet Package

calls the *findPR()* API call. Figure 7.1 and Figure 7.2 only show the resources in one aggregate because all participants are found near the Kentucky Aggregate. The Network Hypervisor chose a random node from the VNIP closest to each participant as that participant's attachment point. This step may take longer in the future as we include more aggregates (VNIPs) in the HyperNet network. We used the algorithm described earlier to build the RP-based multicast tree. More precisely, we use the *buildRPTree()* HyperNet Library call to build a RP-based Tree. As shown in the table, the time spent in step "Build Tree" is relatively small because we have a small physical topology (Figure 7.1). However, we expect that even in a larger topology, the time spent in building an RP-tree will not grow significantly as the number of nodes increase, because the *buildRPTree* API call can take advantage of the cached shortest paths between leave nodes in order to find the rendezvous point and then build the tree. "Deploy Network" takes most of the overall time for each experiment, consuming about 100 seconds. For our experimental testbed, this step includes reserving nodes and links from protoGENI, starting up reserved nodes, uploading *pimd* daemon software (1.4 Megabytes) onto each reserved node, configuring *pimd* properly on each programmable node and executing *pimd*. Interestingly, unlike the other steps, the amount of time spent in this step may vary by as much as 20 seconds across experiments. Since this step is carried out by ProtoGENI, the hypervisor has no control over it. Finally, the average time it takes for a participant to join this multicast network is about 0.7 second. This step is measured from the HyperNet End System package sending out a join request until the end system receives the tunnel information (including instructions for setting up a GRE tunnel). To test the correctness and performance of the multicast tree, we compared the loss rate experienced by each receiver under different sending rates in cases when the sender was multicasting and when the sender was sending out multiple unicast UDP flows. Table 7.2 shows the performance results for multicast vs multi-unicast. Each link in

Table 7.2: Network Performance of Multicast versus Multiple Unicast

	Sending Rate	Loss Rate on each Receiver (%)				
		R1	R2	R3	R4	R5
Multi-cast	20Mbps	0	0	0	0	0
	50Mbps	0.033	0	0	0	0
	90Mbps	0.12	0.01	0.34	0.036	0.049
Uni-cast	20Mbps	15	0.073	0.062	6	5.7
	50Mbps	66	75	58	36	76
	90Mbps	76	79	76	83	82

the reserved network had a bandwidth of 100Mbps, and the bandwidth of the GRE tunnel between each participant and its attachment point was 1Gbps. As expected, in the multicast case, the receivers only experienced minor loss rates even when the sender is sending out packets at a rate of 90Mbps, which is close to the full capacity of the network (100Mbps). However, in the multiple unicast case, the sending rate from the sender to each of the 5 receivers does not exceed 20Mbps for "reliable" delivery (i.e., with minor loss rate experienced by the receivers). In all of our tests we used *iperf* as the network performance testing tool. All results were generated by *iperf* clients on the receivers.

7.2 A MobileNet HyperNet

Mobile devices are increasingly being used for data intensive applications such as browsing web pages, watching videos, streaming music, downloading files, and installing updates. Because these applications typically run over TCP, performance can be seriously degraded when the first hop is a lossy or intermittent wireless link, as is often the cases with mobile devices [86]. Even a loss rate of 0.1% can significantly affect TCP's performance. MobileNet addresses this problem by using a well-known technique called TCP splitting [87] that breaks the TCP connection into two parts: one TCP connection traversing the wireless link, and another TCP connection traversing the wired portion of the path. Consider the network shown

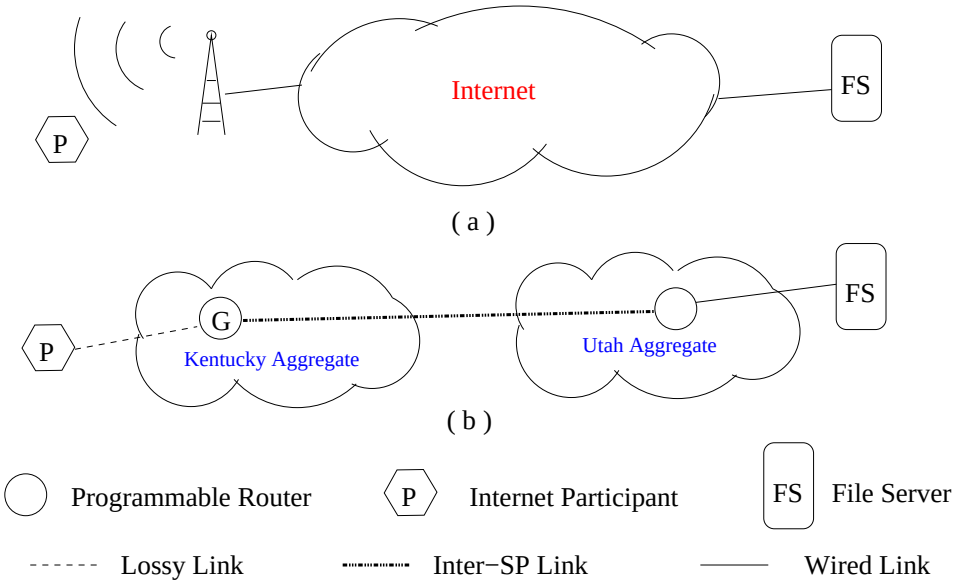


Figure 7.3: A MobileNet HyperNet

in Figure 7.3(a) in which wireless participant P wants to download data from a web server S. Because P's first hop is a wireless link, even small packet losses over the wireless link can severely degrade TCP throughput. Figure 7.3(b) shows the virtual network topology created by MobileNet. The MobileNet HyperNet Package is designed to load special TCP splitting software onto node G to intercept TCP communication between P and S in an attempt to improve TCP performance. The key point is that MobileNet needs to find a programmable router G that is close to P where the TCP splitting software will be run.

Initially P (voluntarily) joins the virtual network resulting in an "'upcall" *join()* being made to the MobileNet HyperNet Package. By using the *findPR()* API call, MobileNet finds a programmable router, G, near P.

MobileNet is an interesting example because it illustrates the need for involuntary participants. The server S is a standard web server and will not, on its own accord, join P's HyperNet network. Consequently, P must make an involuntary join request to join S to its network. Thus, P invokes an involuntary join for server S, which causes an "upcall" to MobileNet to find a programmable router H near S and to

set up H as the proxy (jumping-off point) to S. In our experiment S is attached to the Utah aggregate, so the best H is found to be in the Utah GENI Aggregate. Next, the HyperNet creates a transparent tunnel connecting G to the jumping-off point H using the *addTunnel()* API call. The *addTunnel()* call identifies the two aggregates containing G and H, and then talks to the aggregate managers to setup a GRE tunnel as the transparent tunnel, with routing tables properly set on both ends. A TCP splitting application (we use a script that leverages "netcat" [88]) is then loaded onto G using the *loadApp()* API call. Whenever P communicates with S, this TCP splitting application intercepts TCP packets and improves the end-to-end throughput.

7.2.1 MobileNet Results

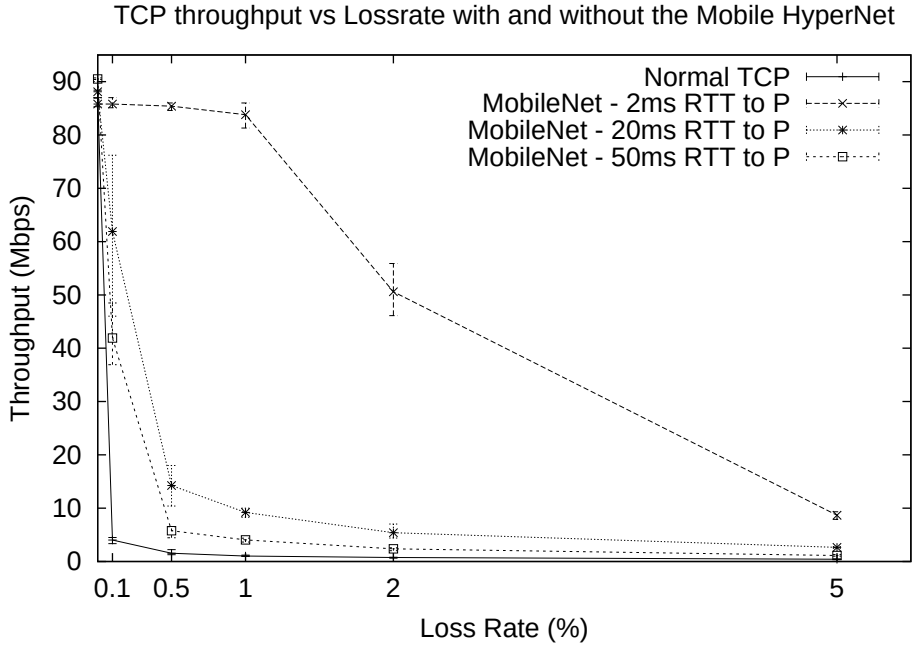


Figure 7.4: MobileNet vs normal TCP Performance

In our experiment, G is found to be a PC in the (wired) Kentucky GENI Aggregate capable of acting as P's attachment point in to the virtual network and also as the TCP-splitting point. H is found to be a PC in the Utah Aggregate acting as the

jumping off point PR for S since S is a web server we created using a PC in Utah Aggregate.

Figure 7.4 shows the performance result of our MobileNet virtual network, compared with regular TCP's performance. In all of our experiments, we use a fixed Round-Trip Time of 100 ms between P and S. We used the traffic control toolkit "tc" in Linux to control the delay and loss rate of each link. In particular we varied the loss rate and delay on the first-hop link between P and G. To evaluate normal TCP performance, traffic simply went between P and S without going through G and H. For the TCP splitting tests, all traffic went through G and H. To understand how much the delay of the wireless link between P and G affects MobileNet performance, we used RTTs between P and G of 2ms, 20ms, and 50ms in our MobileNet tests, and we varied the loss rate between 0% and 5%.

Looking at the normal TCP curve (the lowest curve on the graph), we see that TCP performance decreases rapidly, going from 90 Mbps at a 0% loss rate to 4 Mbps at a 0.1% loss rate and then quickly dropping toward 0 Mbps.

MobileNet (with a 2ms RTT between P and G) set up a split TCP connection that was able to overcome small amounts of packet loss. The top curve shows that even at loss rates up to 1%, MobileNet is able to maintain throughputs around 85 Mbps. As expected, moving G away from P (i.e., increasing the RTT between them) reduces the throughput, but it is still better than normal TCP performance in all cases.

7.3 A Multiplayer Gaming HyperNet

The multiplayer on line gaming HyperNet Package we created aims to shorten the network latency between the players and the game server thereby improving the gaming experience for all players. In this example, we created a gaming hypernet that automatically deploys a custom virtual network for the OpenArena game [89].

OpenArena is an open source multiplayer online game. The game has several publicly accessible game servers to which participants can connect. However, being an open source game, it also allows participants to compile and run their own game servers. Consequently, it is possible for a set of players to identify the best location for a game server (e.g., a location with the lowest delay for all participants), and then run their own game server at that location.

The goal of our OpenArena Gaming hypernet was to automatically select the best location for a game server, run the game server at that location, and then set up an virtual network to connect the game server to all the participants.

Algorithm 3: Multiplayer Gaming HyperNet Package

```
myHyperNet = readConfig() //Read configuration file
regHyperNet(myHyperNet) //Register HyperNet network
while participantNum <predefinedtotalNum do
   joinRequest = checkJoin() //get join request
   Participant p = new Participant(joinRequest)
   myHyperNet.participantList.put(p) //add p onto participant list
   participantNum++
end while
for every participant p in myHyperNet do
   gateway = findPR(p)
   gatewayList.add(gateway)
   participantGWMap[p] = gateway
end for
//find a central node where we can run a game server
gameServer = findCentralNode(gatewayList)
for every gateway g in gatewayList do
   addTunnel(g, gameServer) //create a virtual link
end for
//load Open Arena game server software onto the central node
loadApp(gameServer, OpenArenaServer)
buildTopo() //deploy the topology
Start the Open Arena Server
//connect all participants to the game network
for every participant p in myHyperNet do
   tunnelInfo = new TunnelInfo(p, participantGWMap[p])
   addGW(tunnelInfo)
end for
```

The pseudo code for the OpenArena [89] Gaming HyperNet Package is illustrated

in Algorithm 3. The HyperNet first reads the configuration file for the HyperNet network, including the name of the HyperNet network and the password needed to join the network. Then it registers the HyperNet network using the *regHyperNet()* API call. Then the HyperNet waits for join requests from game players. Whenever a player joins the HyperNet network, the Gaming HyperNet Package keeps a record of the player's IP address. A player's IP address can be found from the *join()* API call. When all players have joined (this is decided by checking the total number of players who have already joined the game), the HyperNet Package then finds a nearby attachment point router for each of the players via the *findPR()* API call. Next, it finds a central node that provides the minimum average delay to all the gateways via the *findCentralNode()* API call, which takes the list of participant (player) attachment points as a parameter. It then creates a virtual channel from each attachment point to the central node. The HyperNet Package then loads the OpenArena server application on the central node via the *loadApp()* API call[1]. Finally the HyperNet Package deploys the game network using the *build()* API call and then starts the OpenArena game server with the following command on the central node:

```
./oa_ded.i386 +set dedicated 1
     +exec server.cfg
     +set net_ip 10.128.2.2
     +set net_port 27961
```

The *+set dedicated 1* parameter means this dedicated server is not visible to the OpenArena master server. In other words, other Internet players are not able to see this server from their in-game *Internet Server* list. It is dedicated to only the three players in our game. The *+exec server.cfg* parameter means the server is launched based on the configuration file *server.cfg*. The *server.cfg* file defines the

[1]In our experiment, we were using OpenArena server version 0.8.8, with the actual tar-ball size of 447 MB. It took about 48 seconds to load the application to the server node.

game parameters such as server name, the maximum number of clients allowed, the maximum sending rate, the maximum allowed ping time from the clients, the game types(e.g., tournament, team death match, last man standing, etc), and the game map. The HyperNet carefully configures this file to optimize the gaming experience for the players. The *+set net_ip* and *+set net_port* parameters simply define the listening interface and listening port number on the server. Again, the HyperNet is responsible for finalizing these parameters to make sure that the game server will not be blocked by firewalls or other networking issues.

At this point, the game virtual network is up and running. Next, the HyperNet Package figures out the configuration for the corresponding gateways (IP addresses that need to be assigned, and the routing table entries that need to be added, etc) for each game player (participant). It then creates a GRE tunnel between each player and its assigned gateway using the *addGW()* API call. In our case, the HyperNet Package reserves 100Mbps bandwidth for all virtual links shown in Figure 7.5, which is more than sufficient for the maximum game sending rate of 200 Kbps set in server.cfg.

The HyperNet network is fully expandable to allow dynamic game joiners, i.e., by finding and assigning more gateways to new requesting game players. Moreover, it can also be designed to locate new game servers according to the current players and migrate all gaming data from the old server to the new one.

7.3.1 Multiplayer Game Results

In our experiment, we used two VNIPs: the Kentucky Aggregate and the Utah Aggregate. Three game players are involved: two are located in Kentucky, the other one is from Utah. As a result, two attachment point PRs are found from Kentucky Aggregate and one attachment point PR is found from Utah Aggregate for the player in Utah. We implemented the Game HyperNet Package and compared it with the standard OpenArena game using existing public game servers. Using the Network

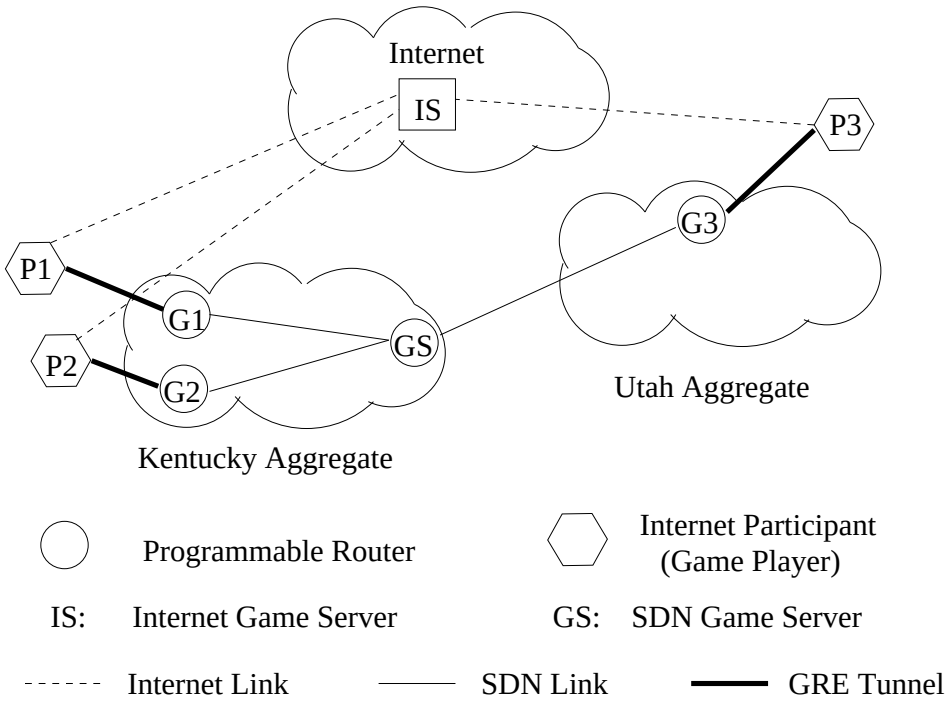


Figure 7.5: Multiplayer Gaming HyperNet

Hypervisor API calls, it only took about 120 lines of Java to encode our OpenArena game HyperNet Package. We then ran our OpenArena game HyperNet Package on the Network Hypervisor to create a custom game network specifically designed for our three players. As shown in Figure 7.5, we were able to find an optimal game server (GS) in ProtoGENI that provided an average round trip time of 22 ms to the three participants. In comparison, the best public Internet Server ("IS" in the figure) provided an average round trip time of 212 ms, showing that custom placement of the game server can provide an order of magnitude improvement in the game's response time. Moreover, by using dedicated lines between the game server and the attachment point PRs, our HyperNet game network also improves the reliability of the game play.

7.4 An OpenFlow Load-balancing HyperNet

To show that our HyperNet Architecture supports the creation of OpenFlow Networks, we created an OpenFlow Load Balancing HyperNet Package (or OFLBH)

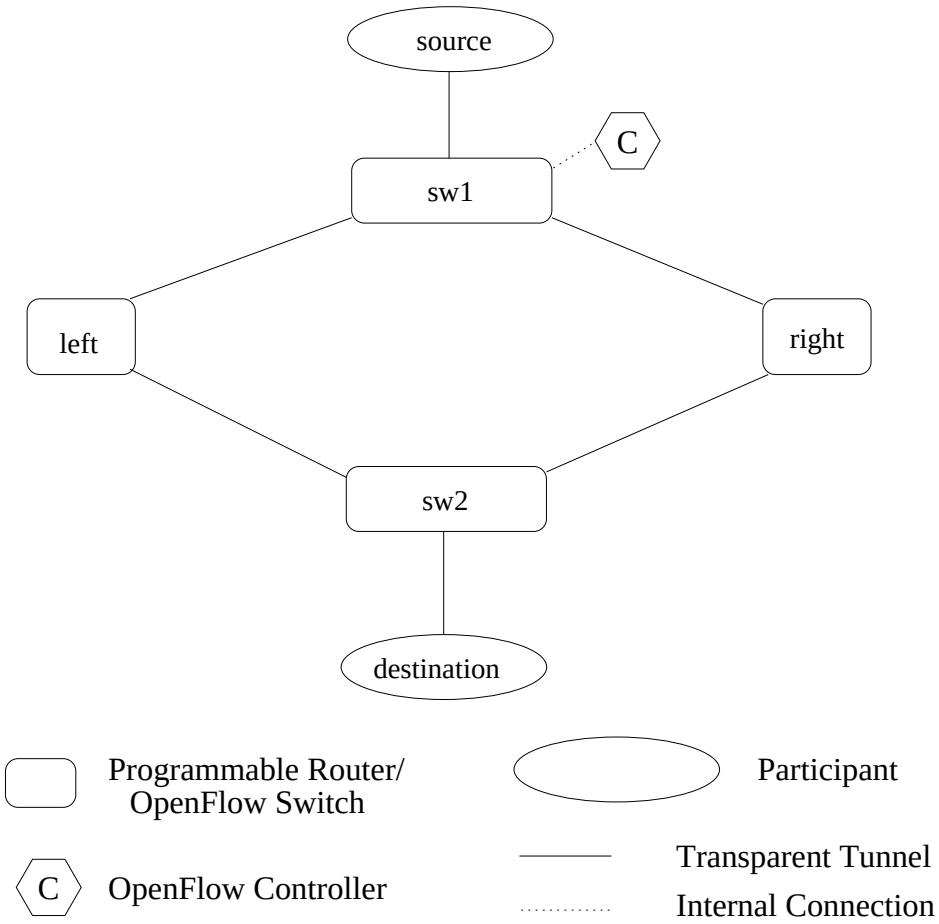


Figure 7.6: OpenFlow Load Balancing Topology

that deploys an OpenFlow load balancing network between two end systems using OpenFlow switches to balance load in ExoGENI. Like ProtoGENI, ExoGENI is another control framework provided by GENI that supports the GENI AM API.

The OFLBH package reserves two paths between two nodes called "source" and "destination", as shown in Figure 7.6. Two OpenFlow switches called "sw1" and "sw2" are used in the topology to load-balance packets. We used a customized Linux kernel with Open vSwitch (a software version of an OpenFlow switch that supports the OpenFlow standards) pre-installed as the operating system on the two OpenFlow switches. As a result, node "sw1" and node "sw2" understand OpenFlow protocols and act like physical OpenFlow switches. The HyperNet Package achieves this by setting the "image" property of each OpenFlow node to load a customized OVS

(Open vSwitch) kernel.

The other two nodes "left" and "right" are simply two relay nodes that forward packets between "sw1" and "sw2". A load-balancing OpenFlow Controller is loaded on node "sw1", which makes the path decision upon arrival of a new traffic flow.

Upon receiving the first packet of a flow from node "source", node "sw1" will forward that packet to the load-balancing OpenFlow controller that is in charge of deciding which path ("left" or "right") to forward the packet over. After making the path decision, the controller pushes the corresponding flow entries into "sw1", so that the node "sw1" will forward all future packets from the same flow over the same path. The OpenFlow Switch "sw2" simply follows whatever decision is made by the controller so that all reverse packets (e.g., the ACK packets from the same TCP flow) will follow the reverse path of the incoming flow.

The HyperNet Package provides two kinds of load balancers (OpenFlow Load Balancing Controllers) for a Network Creator to choose from. Load-balancer A makes path decisions based on the number of flows sent out to each path — it always forwards a new incoming flow to the path with the fewest active flows (essentially alternating between the left and right paths in our example topology). Load-balancer B makes path decisions based on the average per-flow throughput — it always forwards a new incoming flow to the path with higher average per-flow throughput, since, for TCP flows, a higher throughput means that the network path is less congested compared to a flow with lower throughput.

7.4.1 Load Balancing Results

In our experiment, node "source" and node "destination" are two infrastructure participants in the ExoGENI BBN Aggregate. As a result, all other nodes in Figure 7.6 are programmable routers in the same Aggregate. After the OFLBH network was deployed, we started a TCP flow using *iperf* [90] from node "source" to

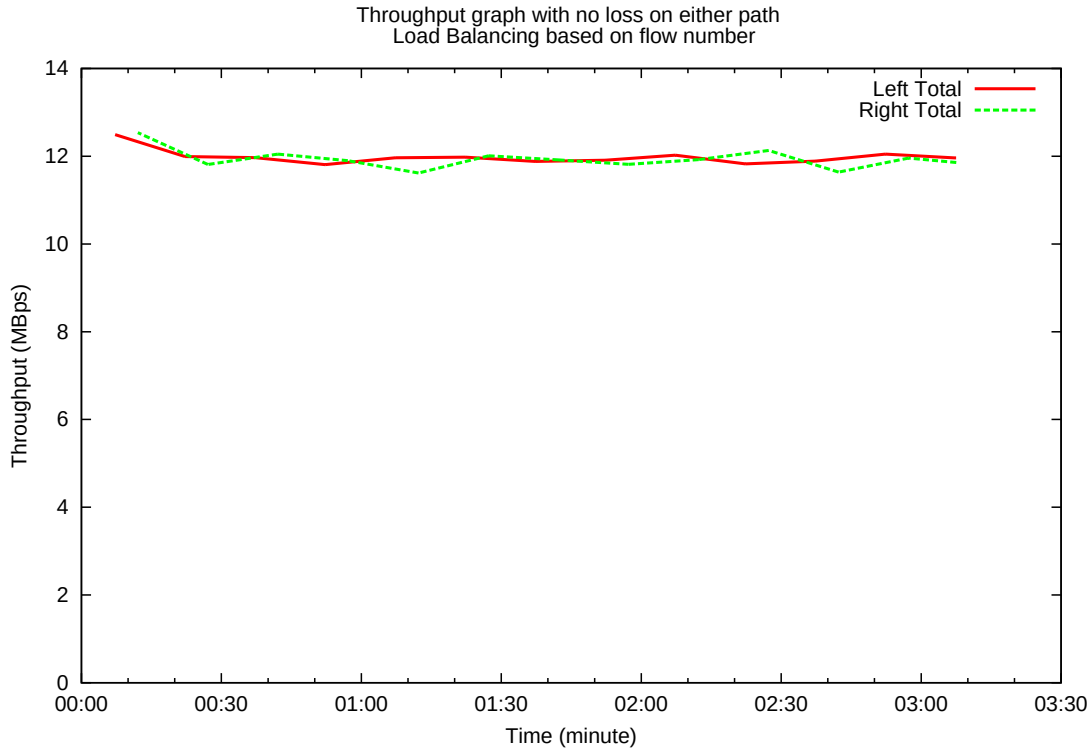


Figure 7.7: Load Balancer A total performance with no loss on either path.

node "destination" every 5 seconds, with a total number of 20 individual TCP flows. We name each individual flow such that flow 1 starts 5 seconds earlier than flow 2, etc. All flows last for 200 seconds. We measured both the individual throughput of each flow as well as the total throughput on each path (left and right).

Figure 7.7 shows the total throughput on left and right paths under no loss using Load Balancer A. As we can see, the load balancer makes full use of the bandwidth on both paths (about 12MBps or 100Mbps).

Figure 7.8 shows the individual flow throughput using load-balancer A when there is no loss on both left and right paths. From the per-flow performance graph we can see that for both left and right paths, the throughput of each flow decreases as new flows arrive. This result is due to TCP's congestion control mechanism: as new flows arrive, the network starts to drop packets and TCP automatically adjusts its congestion window size (and thus, throughput) for each flow. For the duration of

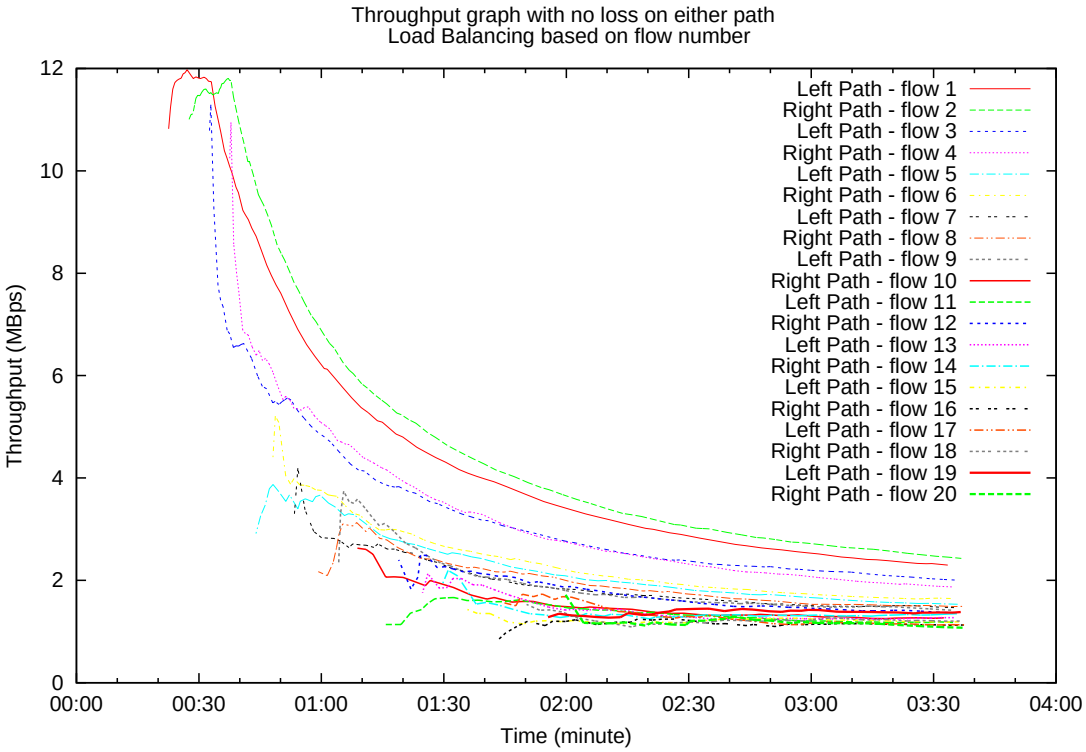


Figure 7.8: Load Balancer A per-flow performance with no loss on either path.

our experiments (about 200 seconds) we can see that the throughput of all TCP flows gradually dropped to around 2MBps. TCP's congestion control mechanism will eventually balance the sharing of bandwidth such that each TCP flow will use similar amount of bandwidth.

The per-flow throughput and total throughput performance results for load-balancer B is similar to the results for load-balancer A when there is no loss for both paths and thus, we omit the graphs here.

Figures 7.9 and 7.10 show the total performance as well as the per-flow performance on both paths using load-balancer A when there is 5% loss rate on the left path. Due to A's load balancing algorithm, despite the fact that there is 5% loss rate on the left path, the OpenFlow controller still forwards the new incoming TCP flows in an alternating pattern onto left and right paths. The result is, 10 flows are forwarded to the left path and 10 on the right path. The flows on the left path

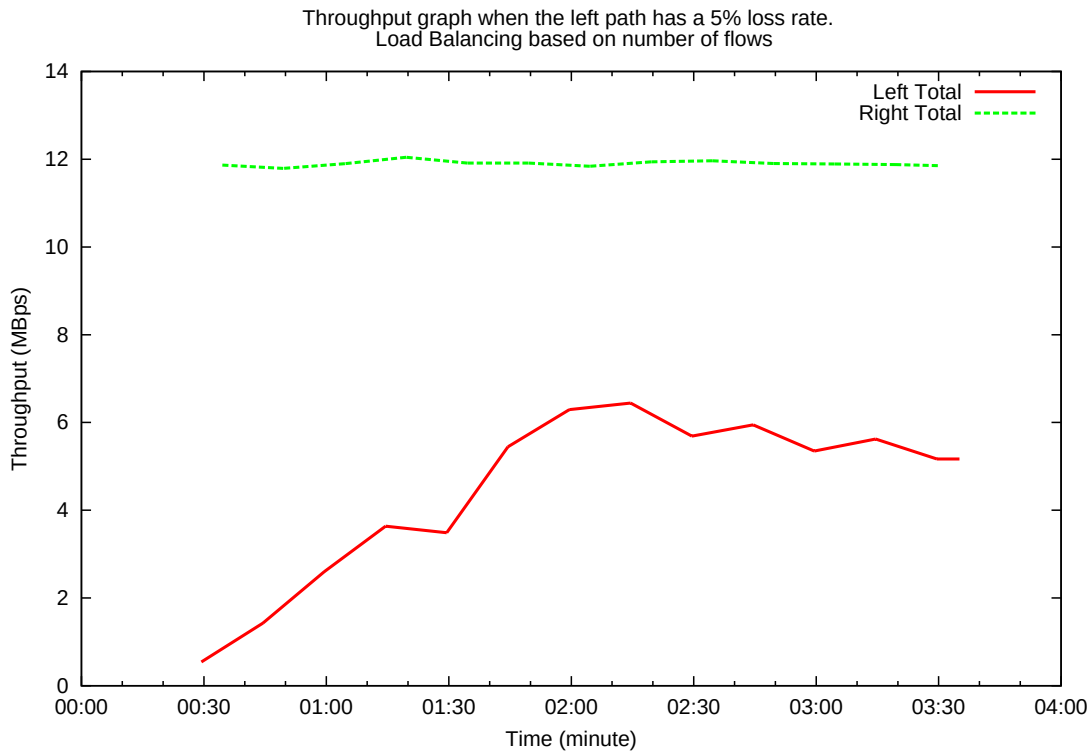


Figure 7.9: Load Balancer A total performance with 5% loss on the left path.

achieve very low throughput compared with the throughput of flows on the right path.

On the other hand, if we use the load-balancer B, which makes load balancing decisions based on the measured average per-flow throughput, we see that only TCP flow 1 was directed to the left path from Figure 7.12. Because of the losses on the left path, the average per-flow throughput is much lower on the left path than the right path. Therefore no more flows are assigned to the left path after flow 1. TCP flow 1 was able to achieve a throughput of 0.5MBps. The remaining 19 TCP flows were forwarded to the right path, sharing the total available 12MBps bandwidth. If we compare Figures 7.9 and 7.11, we can see that in the case of 5% loss on the left path, load-balancer A achieved a higher overall throughput between "outside" and "inside" (the sum of total throughput on left and right paths) than load-balancer B. However, when comparing Figures 7.10 and 7.12, we see that load-balancer B provides better

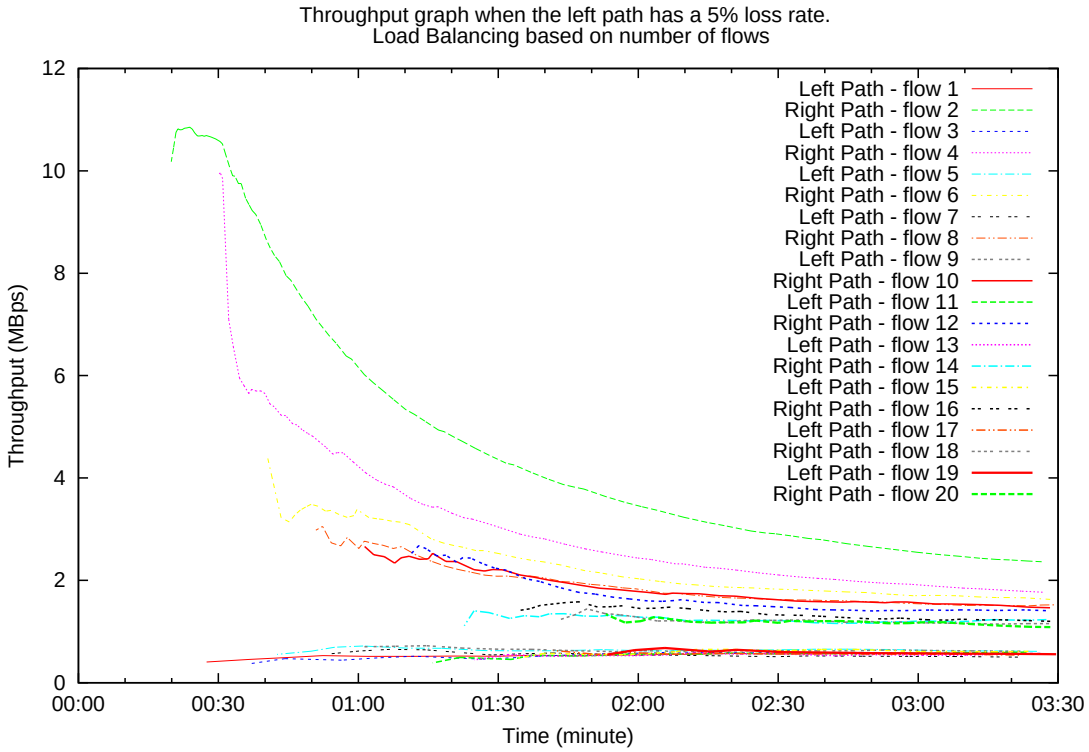


Figure 7.10: Load Balancer A per-flow performance with 5% loss on the left path.

fairness to all TCP flows going through the load balancing network in the sense that almost all TCP flows achieved similar throughput; whereas using load-balancer A, the flows on the right path achieve much higher bandwidth than flows on the left path. As a result, depending on the different characteristics of each path (loss rate) and the virtual network users' needs, the Network Creator might want to choose a different kind of load-balancer to handle the load balancing task.

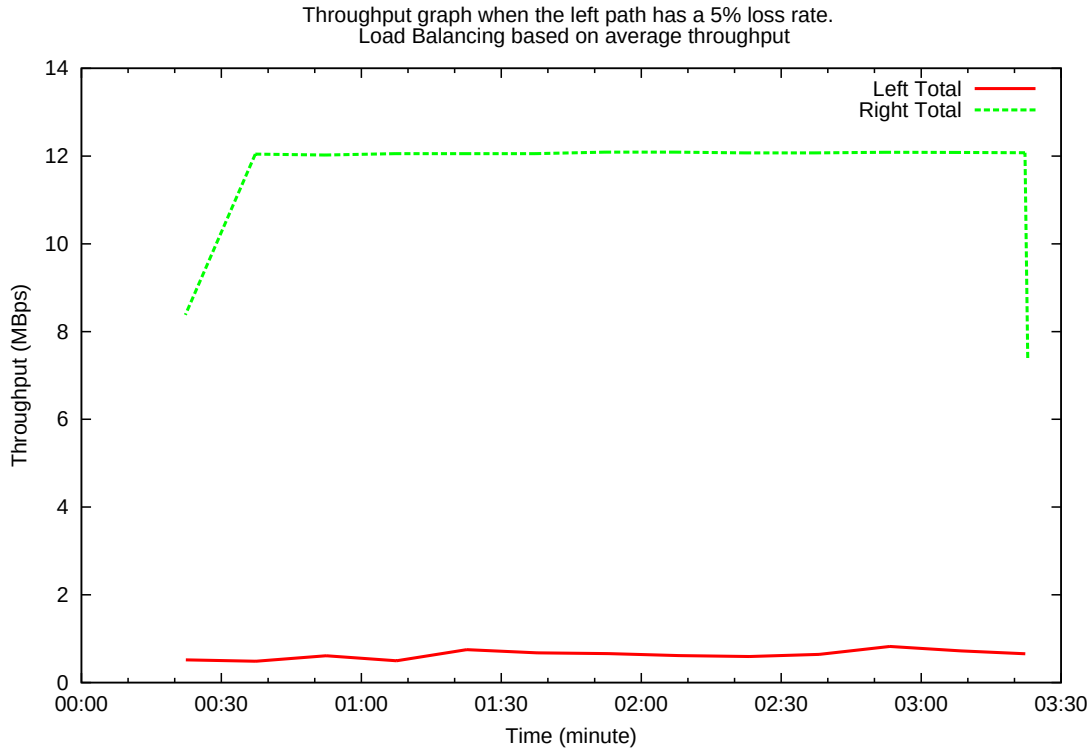


Figure 7.11: Load Balancer B per-flow performance with 5% loss on the left path.

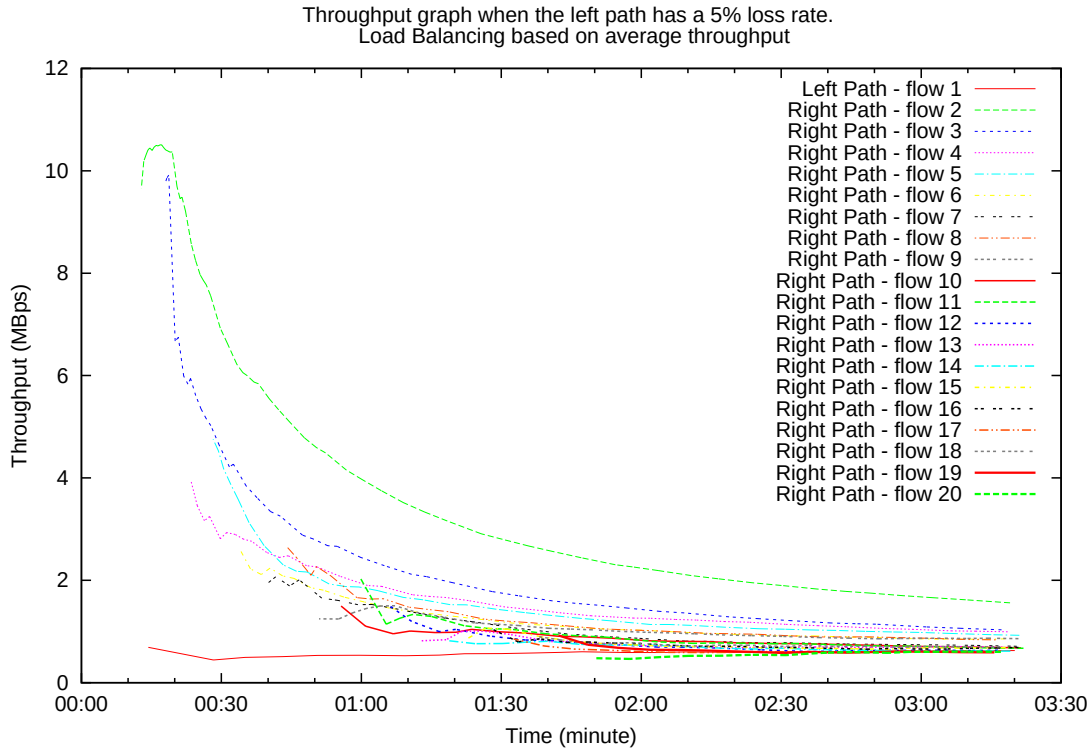


Figure 7.12: Load Balancer B per-flow performance with 5% loss on the left path.

Chapter 8

Conclusion and Future Work

In this thesis, we introduce the concept of a *HyperNet*. Modeled after a virtual appliance, a HyperNet is a software package that contains all the necessary pieces needed to create a special-purpose virtual network, including topology specifications, protocol stacks, software packages, configuration files, and runtime scripts, so that all that one needs to do to deploy a virtual network is to download a HyperNet from the HyperNet market and "run" it. The HyperNet abstraction makes it possible (and extremely easy) for an average user to create a virtual network, which otherwise would be a challenging and labor-intensive task even for a network expert.

We introduce a HyperNet Architecture that supports the HyperNet abstraction. We also define the concept of a Virtual Network Infrastructure Provider (VNIP) which provides virtual networking resources to users for a fee. We describe our assumptions about VNIPs and the common set of services provided by VNIPs. We also show how these services map onto the services provided by today's existing virtual network testbed providers (e.g., GENI).

At the heart of the HyperNet architecture is the Network Hypervisor, which provides the platform for HyperNets to run on. The Network Hypervisor acts as the broker between HyperNet users (i.e., Network Creators and HyperNet Participants) and VNIPs. The Network Hypervisor talks to VNIPs on behalf of HyperNet users with different VNIPs to reserve resources, and it glues the resources together to form

a virtual network that might span multiple VNIPs. The Network Hypervisor provides a set of Hypervisor API calls that HyperNet Builders can use to build a wide range of HyperNets.

To demonstrate our design, we implement a Network Hypervisor using GENI as the underlying VNIP. The Network Hypervisor contains multiple components that help (1) maintain an up-to-date resource information of the underlying VNIPs, (2) locate network nodes, including end systems (i.e., HyperNet Participants), (3) calculate paths between network nodes, and (4) support the Network Hypervisor APIs. Our implementation is extensible in the sense that any third party developer can contribute by implementing "HyperNet Libraries" based on the provided Network Hypervisor APIs. We implement a "HyperNet Topology Library" as well as a "HyperNet Routing Library" to ease the task of creating virtual networks having common shape and to automatically manage the routing tables of the nodes in a virtual network. Experimental results show that the Network Hypervisor is able to quickly create a virtual network topology and can scale as the number of HyperNets increases, particularly since the Network Hypervisor can be parallelized.

To showcase the power of the HyperNet architecture, we created four HyperNet examples. Performance results show that each of the virtual networks provides functionality not present in the current Internet or out-performs what the Internet can offer. The Multicast HyperNet enables the user to create a multicast network over the Internet and thus effectively use the network bandwidth via multicasting. The MobileNet example enables a mobile user to achieve 50Mbps throughput with up to 5% network loss rate. The Multiplayer Gaming HyperNet drastically shortens the average RTT from game players to the centralized game server. The OpenFlow Load Balancing HyperNet enables the user to create multiple paths between source and destination and to intelligently load balance network traffic going through each path based on the network condition of each path. The point is not that these special-

purpose networks are particularly novel or new, but rather that they can be created with the HyperNet architecture, and that they can be easily created and deployed. With the help of the Network Hypervisor APIs and the HyperNet Libraries that we implemented, it only takes around 200 lines of Java code to implement each of the HyperNet examples. More importantly, to deploy any of those virtual networks, one only needs to download the HyperNet Package and execute it.

8.1 Future Work

Our HyperNet system shows great promise as a way to deploy and manage special-purpose virtual networks, but there remain aspects of the architecture that need additional study.

The following outlines next steps that would help to bring about a complete HyperNet architecture:

Devise a business model for the HyperNet architecture

The emergence of VNIP providers will definitely break the current business model used by existing ISPs. A thorough mechanism/model for how VNIPs charge Hypervisors, which in turn charge HyperNet users, needs to be designed. It is also possible for network creators to offer their specialized virtual networks for a fee to the HyperNet participants. New models that monitor networking resource usage need to be designed so as to properly charge all types of users in the HyperNet architecture.

Incorporate an "update" API call in the Network Hypervisor

Dynamic joining and leaving of participants may lead to the need to update the virtual topology, or a Network Creator might want to renegotiate an allocation. An *updateTopo()* API call should be provided by the Network Hypervisor. This API call should be implemented such that an "update" operation on the virtual

network interferes as little as possible with the communication between existing participants. VNIPs might allow network updates and implement it efficiently. GENI actually provides very limited support for an "update" operation on an existing slice.

We believe the HyperNet Package has information about the current virtual network and will know if it is safe to update the network. Consequently, it can be designed to apply non-interfering changes to the network (e.g., updating routing table entries as little as possible) upon the addition or removal of a new programmable router or a new virtual link.

Runtime control for the network creator

Although the HyperNet abstraction was designed for an average user to create special-purpose virtual networks with no or trivial effort (e.g., modifying a simple HyperNet configuration file), in this thesis we did not explore any runtime interactions between the HyperNet package and the network creator after a virtual network is created. Advanced network creators might want to manually interact with the virtual network. For example, they might want to assign HyperNet-specific addresses to individual participants, or to manage participants' memberships (e.g., assigning "managers" who can add/remove participants), or to pause/stop/restart/save/restore a virtual network. Additional Network Hypervisor API calls need to be designed to support such functionalities. A Network Creator is an average user, so a GUI running on the Network Creator's end system or a web interface provided by the Network Hypervisor could provide a helpful, easy-to-use, interface for a network creator to control the network at runtime.

QoS Support

Although the most important contribution of HyperNets is that it enables

ordinary users to run their own special-purpose virtual networks, many of the envisioned HyperNets will need QoS support to be useful. As a result, to make HyperNets more attractive, the Network Hypervisor should offer the ability to provision resources to ensure they can meet the desired performance requirements. The API used by the HyperNet Builder might specify QoS requirements on a per-tunnel or overall basis, and if the VNIP does not support such provisioning, there must be a standard fallback technique. Either granted by the underlying VNIP or by the Network Hypervisor, new models and algorithms are needed to offer QoS support.

Robust Fault Handling

Any robust system must be designed to deal with failures. The same holds true for the HyperNet architecture. In our architecture, failures may happen in a VNIP (e.g., a physical node/virtual node fails, a physical link breaks, or a virtual link disconnects), in a Network Hypervisor, in a HyperNet participant (e.g., a participant accidentally loses connection with its assigned attachment point and wants to reconnect to the network). For some of the failures, the HyperNet architecture might be able to automatically and seamlessly recover from them without being noticed by the users. For example, a VNIP may implement mechanisms to migrate a failed virtual router to a nearby healthy programmable router while maintaining all the running states of the previous router at the moment of failure. Similarly, the hypervisor "cloud" might include features/mechanisms to automatically "redirect" users' requests to healthy nearby hypervisor instances in the case of a hypervisor instance failure.

Security

Security is not covered in this thesis, but mechanisms to deal with potential security threats in the HyperNet architecture need to be designed to ensure a

healthy system welcomed by the users. Since anyone can download any number of HyperNet Packages and deploy them into HyperNet networks, the Network Hypervisor might implement mechanisms to prevent a Network Creator from consuming all VNIP resources from other Network Creators. Moreover, since the same HyperNet Package can be downloaded by potentially a large number of Network Creators and be deployed into many HyperNet networks, a certain trust relationship needs to be built between a HyperNet Builder and the HyperNet Market (or certain security check mechanisms need to be built in the HyperNet Market platform) to make sure that a maliciously built HyperNet Package can not be uploaded by a HyperNet Builder (and hence will not be downloaded and deployed by a Network Creator).

Bibliography

[1] A. Anand, F. Dogar, D. Han, B. Li, H. Lim, M. Machado, W. Wu, A. Akella, D. G. Andersen, J. W. Byers, S. Seshan, and P. Steenkiste, "XIA: An Architecture for an Evolvable and Trustworthy Internet," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, ser. HotNets-X. New York, NY, USA: ACM, 2011, pp. 2:1–2:6. [Online]. Available: http://doi.acm.org/10.1145/2070562.2070564

[2] T. Wolf, J. Griffioen, K. L. Calvert, R. Dutta, G. N. Rouskas, I. Baldine, and A. Nagurney, "Choice as A Principle in Network Architecture," *SIGCOMM Comput. Commun. Rev.*, pp. 105–106, Aug. 2012.

[3] F. Bronzino, K. Nagaraja, I. Seskar, and D. Raychaudhuri, "Network Service Abstractions for a Mobility-centric Future Internet Architecture," in *Proceedings of the Eighth ACM International Workshop on Mobility in the Evolving Internet Architecture*, ser. MobiArch '13. New York, NY, USA: ACM, 2013, pp. 5–10. [Online]. Available: http://doi.acm.org/10.1145/2505906.2505908

[4] J. Yang and Z. Fei, "Broadcasting with Prediction and Selective Forwarding in Vehicular Networks," *International Journal of Distributed Sensor Networks*, 2013.

[5] J. Yang and Z. Fei, "HDAR: Hole Detection and Adaptive Geographic Routing for Ad Hoc Networks." in *ICCCN*. IEEE, 2010, pp. 1–6. [Online]. Available: http://dblp.uni-trier.de/db/conf/icccn/icccn2010.html#YangF10a

[6] M. Boucadair, J.-L. Grimault, P. Levis, A. Villefranque, and P. Morand, "Anticipate IPv4 Address Exhaustion: A Critical Challenge for Internet Survival," in *Proceedings of the 2009 First International Conference on Evolving Internet*, ser. INTERNET '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 27–32. [Online]. Available: http://dx.doi.org/10.1109/INTERNET.2009.11

[7] A. Elmokashfi, A. Kvalbein, and C. Dovrolis, "BGP Churn Evolution: a Perspective from the Core," *IEEE/ACM Trans. Netw.*, vol. 20, no. 2, pp. 571–584, Apr. 2012. [Online]. Available: http://dx.doi.org/10.1109/TNET.2011.2168610

[8] T. G. Griffin and G. Wilfong, "An Analysis of BGP Convergence Properties," in *Proceedings of the conference on Applications, technologies, architectures,*

and protocols for computer communication, ser. SIGCOMM '99. New York, NY, USA: ACM, 1999, pp. 277–288. [Online]. Available: http://doi.acm.org/10.1145/316188.316231

[9] R. Mahajan, D. Wetherall, and T. Anderson, "Understanding BGP Misconfiguration," *SIGCOMM Comput. Commun. Rev.*, vol. 32, no. 4, pp. 3–16, Aug. 2002. [Online]. Available: http://doi.acm.org/10.1145/964725.633027

[10] P. Zhang, A. Durresi, and L. Barolli, "A Survey of Internet Mobility," in *Proceedings of the 2009 International Conference on Network-Based Information Systems*, ser. NBIS '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 147–154. [Online]. Available: http://dx.doi.org/10.1109/NBiS.2009.94

[11] S. P. Leblanc, A. Partington, I. Chapman, and M. Bernier, "An Overview of Cyber Attack and Computer Network Operations Simulation," in *Proceedings of the 2011 Military Modeling & Simulation Symposium*, ser. MMS '11. San Diego, CA, USA: Society for Computer Simulation International, 2011, pp. 92–100. [Online]. Available: http://dl.acm.org/citation.cfm?id=2048558.2048572

[12] M. Nicholes and B. Mukherjee, "A Survey of Security Techniques for The Border Gateway Protocol (BGP)," *Commun. Surveys Tuts.*, vol. 11, no. 1, pp. 52–65, Jan. 2009. [Online]. Available: http://dx.doi.org/10.1109/SURV.2009.090105

[13] A. D. Keromytis, "Voice-over-IP Security: Research and Practice," *IEEE Security and Privacy*, vol. 8, no. 2, pp. 76–78, Mar. 2010. [Online]. Available: http://dx.doi.org/10.1109/MSP.2010.87

[14] S. Furnell, "Remote PC Security: Securing The Home Worker," *Netw. Secur.*, vol. 2006, no. 11, pp. 6–12, Nov. 2006. [Online]. Available: http://dx.doi.org/10.1016/S1353-4858(06)70451-2

[15] P. Szewczyk and C. Valli, "Ignorant Experts: Computer and Network Security Support from Internet Service Providers," in *Proceedings of the 2010 Fourth International Conference on Network and System Security*, ser. NSS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 323–327. [Online]. Available: http://dx.doi.org/10.1109/NSS.2010.42

[16] A. Feldmann, "Internet Clean-Slate Design: What and Why?" *SIGCOMM Comput. Commun. Rev.*, vol. 37, pp. 59–64, July 2007. [Online]. Available: http://doi.acm.org/10.1145/1273445.1273453

[17] C. Dovrolis, "What would Darwin Think about Clean-Slate Architectures?" *SIGCOMM Comput. Commun. Rev.*, vol. 38, pp. 29–34, January 2008. [Online]. Available: http://doi.acm.org/10.1145/1341431.1341436

[18] Virtual Appliance. [Online]. Available: http://en.wikipedia.org/wiki/Virtual_appliance

[19] Planetlab. [Online]. Available: http://www.planet-lab.org/

[20] L. Peterson, V. Pai, N. Spring, and A. Bavier, "Using PlanetLab for Network Research: Myths, Realities, and Best Practices," PlanetLab Consortium, Tech. Rep. PDN–05–028, June 2005.

[21] W. D. Laverell, Z. Fei, and J. N. Griffioen, "Isn't it Time You Had an Emulab?" in *Proceedings of the 39th SIGCSE technical symposium on Computer science education*, ser. SIGCSE '08. New York, NY, USA: ACM, 2008, pp. 246–250. [Online]. Available: http://doi.acm.org/10.1145/1352135.1352223

[22] Virtual LAN. [Online]. Available: http://en.wikipedia.org/wiki/Virtual_LAN

[23] GENI, "Global Environment for Network Innovations - System Requirements Document," 2009. [Online]. Available: http://groups.geni.net/geni/wiki/GpoDoc

[24] L. Peterson, S. Sevinc, J. Lepreau, R. Ricci, J. Wroclawski, T. Faber, S. Schwab, and S. Baker, "Slice-Based Facility Architecture," 2009. [Online]. Available: http://www.cs.princeton.edu/~llp/arch_abridged.pdf

[25] GENI. (2009) GENI Research Plan. [Online]. Available: http://groups.geni.net/geni/attachment/wiki/OldGPGDesignDocuments/GDD-06-28.pdf

[26] Juniper, "Juniper M7I Router." [Online]. Available: http://www.juniper.net/customers/support/products/m7i.jsp

[27] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[28] D. Farinacci, T. Li, S. Hanks, D. Meyer, and P. Traina, "Generic Routing Encapsulation (GRE)," RFC 2784 (Proposed Standard), Internet Engineering Task Force, Mar. 2000, updated by RFC 2890. [Online]. Available: http://www.ietf.org/rfc/rfc2784.txt

[29] E. Rosen, A. Viswanathan, and R. Callon, "Multiprotocol Label Switching Architecture," RFC 3031 (Proposed Standard), Internet Engineering Task Force, Jan. 2001, updated by RFC 6178. [Online]. Available: http://www.ietf.org/rfc/rfc3031.txt

[30] B. Fenner, M. Handley, H. Holbrook, and I. Kouvelas, "Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification (Revised)," RFC 4601 (Proposed Standard), Internet Engineering Task Force, Aug. 2006, updated by RFCs 5059, 5796, 6226. [Online]. Available: http://www.ietf.org/rfc/rfc4601.txt

[31] G. Peng, "CDN: Content Distribution Network," Tech. Rep., 2003.

[32] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links," *IEEE/ACM TRANSACTIONS ON NETWORKING*, vol. 5, pp. 756–769, 1997.

[33] Linux VServer. [Online]. Available: http://linux-vserver.org/Welcome_to_Linux-VServer.org

[34] OpenVZ Linux Containers. [Online]. Available: http://openvz.org/Main_Page

[35] Linux Containers. [Online]. Available: http://linuxcontainers.org/

[36] Parallels Desktop. [Online]. Available: http://www.parallels.com/

[37] Microsoft. Windows Virtual PC. [Online]. Available: http://www.microsoft.com/en-us/download/details.aspx?id=3702

[38] Virtual Machine. [Online]. Available: http://en.wikipedia.org/wiki/Virtual_machine

[39] IBM. IBM ZOS. [Online]. Available: http://www-03.ibm.com/systems/z/os/zos/

[40] Citrix. Citrix XenServer. [Online]. Available: http://www.citrix.com/products/xenserver/overview.html

[41] VMware. VMware ESX Hypervisor Architecture. [Online]. Available: http://www.vmware.com/products/esxi-and-esx/overview.html

[42] VMware Workstation. [Online]. Available: http://www.vmware.com/products/workstation/

[43] Oracle. Oracle VirtualBox. [Online]. Available: https://www.virtualbox.org/

[44] Kernel-based Virtual Machine. [Online]. Available: http://www.linux-kvm.org/page/Main_Page

[45] FreeBSD Jails. [Online]. Available: https://wiki.freebsd.org/Jails

[46] Drupal Virtual Appliance. [Online]. Available: http://www.turnkeylinux.org/drupal7

[47] AberdeenGroup. (2009) Business Adoption of Cloud Computing. [Online]. Available: http://aberdeen.com/aberdeen-library/6220/RA-cloud-computing-sustainability.aspx

[48] Google Documents. [Online]. Available: http://docs.google.com

[49] Google App Engine. [Online]. Available: http://code.google.com/appengine/

[50] Amazon EC2. [Online]. Available: http://aws.amazon.com/ec2/

[51] OpsCode. (2013) Chef. [Online]. Available: http://www.opscode.com/chef/

[52] "ProtoGENI Network Testbed." [Online]. Available: http://www.protogeni.net/trac/protogeni

[53] I. Baldine and J. Chase, "Deploying a Vertically Integrated GENI "Island": A Prototype GENI Control Plane (ORCA) for a Metro-Scale Optical Testbed (BEN)," *http://groups.geni.net/geni/wiki/ORCABEN*.

[54] GENI-ORBIT. [Online]. Available: http://groups.geni.net/geni/wiki/ORBIT

[55] GENI Flack Web Interface. [Online]. Available: http://www.protogeni.net/ProtoGeni/wiki/FlackTutorial

[56] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM*, 2008.

[57] ORCA-Flukes. [Online]. Available: https://geni-orca.renci.org/trac/wiki/flukes

[58] J. Albrecht, C. Tuttle, R. Braud, D. Dao, N. Topilski, A. C. Snoeren, and A. Vahdat, "Distributed application configuration, management, and visualization with plush," *ACM Trans. Internet Technol.*, vol. 11, no. 2, pp. 6:1–6:41, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2049656.2049658

[59] Planetlab Experiment Manager. [Online]. Available: http://research.cs.washington.edu/networking/cplane/

[60] Planetlab Application Manager. [Online]. Available: http://appmanager.berkeley.intel-research.net/

[61] J. Albrecht and D. Y. Huang, "Managing Distributed Applications using Gush," 2010.

[62] J. Hartman and S. Baker, "Raven Provisioning Tool." [Online]. Available: http://groups.geni.net/geni/wiki/ProvisioningService

[63] Stork Installation Utility. [Online]. Available: http://www.cs.arizona.edu/stork/

[64] D. J. Wetherall, J. V. Guttag, and D. L. Tennenhouse, "ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols," in *IEEE OPENARCH*, 1998.

[65] S. R. Srinivasan, J. W. Lee, E. Liu, M. Kester, H. Schulzrinne, V. Hilt, S. Seetharaman, and A. Khan, "NetServ: Dynamically Deploying In-Network Services," in *Proceedings of the 2009 workshop on Re-architecting the internet*, ser. ReArch '09. New York, NY, USA: ACM, 2009, pp. 37–42. [Online]. Available: http://doi.acm.org/10.1145/1658978.1658988

[66] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, "The Click Modular Router," *SIGOPS Oper. Syst. Rev.*, vol. 33, no. 5, pp. 217–231, 1999.

[67] OSGi Technology. [Online]. Available: http://www.osgi.org/

[68] T. Wolf, "Service-Centric End-to-End Abstractions in Next-Generation Networks," in *Proc. of Fifteenth IEEE International Conference on Computer Communications and Networks (ICCCN)*, Arlington, VA, Oct. 2006, pp. 79–86.

[69] S. Ganapathy and T. Wolf, "Design of a Network Service Architecture," in *Proc. of Sixteenth IEEE International Conference on Computer Communications and Networks (ICCCN)*, Honolulu, HI, Aug. 2007.

[70] N. C. Hutchinson and L. L. Peterson, "The x-Kernel: An Architecture for Implementing Network Protocols," *IEEE Transactions on Software Engineering*, vol. 17, pp. 64–76, 1991.

[71] K. Calvert, "Beyond Layering: Modularity Considerations for Protocol Architectures," *International conference on network protocols*, pp. 90–97, 1993.

[72] R. Clayton and K. Calvert, "Structuring Protocols with Data Streams," *Second Workshop on High-Performance Protocol Architectures*, 1995.

[73] R. Dutta, G. N. Rouskas, I. Baldine, A. Bragg, and D. Stevenson, "The SILO Architecture for Services Integration, Control, and Optimization for the Future Internet," in *IEEE ICC*, 2007, pp. 24–27.

[74] M. Vellala, A. Wang, G. Rouskas, R. Dutta, I. Baldine, and D. Stevenson, "A Composition Algorithm for the SILO Cross-Layer Optimization Service Architecture," *Proceedings of the Advanced Networks and Telecommunications Systems Conference (ANTS 2007)*, 2007.

[75] I. Houidi, W. Louati, D. Zeghlache, and S. Baucke, "Virtual Resource Description and Clustering for Virtual Network Discovery," *Communications Workshops, 2009. ICC Workshops 2009. IEEE International Conference on*, pp. 1 –6, jun. 2009.

[76] "GENI AM API." [Online]. Available: http://groups.geni.net/geni/wiki/GAPI_AM_API_V3

[77] VMware, "Vmware Virtual Appliances," *http://www.vmware.com/appliances/*.

[78] MPEG Video Standards. [Online]. Available: http://mpeg.chiariglione.org/

[79] L. Krishnamurthy, "AQUA: An Adaptive Quality of Service Architecture for Distributed Multimedia Applications," 1997. [Online]. Available: http://search.proquest.com/pqdtft/docview/304385403/14229056CF46A5EE93F/1?accountid=11836

[80] G. Schaffrath, C. Werle, P. Papadimitriou, A. Feldmann, R. Bless, A. Greenhalgh, A. Wundsam, M. Kind, O. Maennel, and L. Mathy, "Network Virtualization Architecture: Proposal and Initial Prototype," in *VISA '09: Proceedings of the 1st ACM workshop on Virtualized infrastructure systems and architectures*. New York, NY, USA: ACM, 2009, pp. 63–72.

[81] Y. Breitbart, C.-Y. Chan, M. Garofalakis, R. Rastogi, and A. Silberschatz, "Efficiently Monitoring Bandwidth and Latency in IP Networks," in *In Proc. IEEE Infocom*, 2001, pp. 933–942.

[82] Protocol-Independent Multicast Daemon. [Online]. Available: http://manpages.ubuntu.com/manpages/maverick/en/man1/pimd.1.html

[83] TUN/TAP Interface. [Online]. Available: http://en.wikipedia.org/wiki/TUN/TAP

[84] InstaGENI Network Testbed. [Online]. Available: http://groups.geni.net/geni/wiki/INSTAGENI

[85] HyperNet Source Code and Builder Manual. [Online]. Available: http://protocols.netlab.uky.edu/~shufeng/pvn/

[86] H. Balakrishnan, V. N. Padmanabhan, S. Seshan, and R. H. Katz, "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links," *IEEE/ACM TRANSACTIONS ON NETWORKING*, vol. 5, pp. 756–769, 1997.

[87] R. Jain and T. J. Ott, "Design and Implementation of Split TCP in the Linux Kernel," in *GLOBECOM*. IEEE, 2006. [Online]. Available: http://dblp.uni-trier.de/db/conf/globecom/globecom2006nxg.html#JainO06

[88] The GNU Netcat. [Online]. Available: http://netcat.sourceforge.net/

[89] OpenArena Online Shooting Game. [Online]. Available: http://openarena.ws/smfnews.php

[90] The Iperf Network Measuring Tool. [Online]. Available: http://iperf.sourceforge.net/

Vita

Shufeng Huang

- **Education**
  - B.S. in Computer Science, Beijing Normal University, Beijing, China, 2006
- **Research Experience**
  - Designed and implemented a toolkit called "switchspider" that traverses a network of connected switches and fetches information about IP phones using SNMP and CISCO MIB, combined with info fetched from Cisco's CUCM server (via XML-RPC), providing an integrated information base for University of Kentucky's VoIP Phone users.
  - Research Assistant, 2007 – 2013, Department of Computer Science, University of Kentucky
    * Worked in VOEIS project, co-designed and implemented and IOS app and a "dataspoke" system that fetches streaming data from buoys which monitor the environmental variable changes for Kentucky Lake and Flathead Lake in Montana.
    * Worked in PoMo project, co-designed and implemented the forwarding plane as well as the E2L (EID to Locator) service for the PoMo network.
    * Worked in the Treasury Project, designed and implemented a new Remote Backup system with "tracker" in Linux Kernel using RB trees. Co-designed and implemented a Reliable FEC transport protocol that aims to minimize per-packet delay.
  - Intern at Raytheon BBN Technologies, 2013 summer, Boston
    * Build the "Network Hypervisor" platform that facilitates the creation and deployment of virtual networks on GENI (an extension of my thesis work).
    * Created advanced OpenFlow tutorials on building a load balancer and a firewall using Trema.
    * Created advanced Content-Centric Network (CCN) tutorials on exploring features of CCN networks, using CCNX toolkit.
    * Created advanced TCP tutorials on experimenting with the performance of different TCP congestion control algorithms.

- Intern at RockTech, 2005 summer, Beijing, China
  - ∗ Implemented a Demo website displaying SICAD-generated maps using JSP
- Implemented Reed-Solomon algorithm as one of the error correction library in the "sky-fix stone" disk recovery toolkit for Convoy Data Technologies Co. Ltd., Beijing, China

• Publications

- S. Huang, J. Griffioen and K. Calvert, "Network Hypervisors: Enhancing SDN Infrastructure", invited paper in COMCOM Journal, 2014
- S. Huang and J. Griffioen, "Network Hypervisors: Managing the Emerging SDN Chaos", 22nd International Conference on Computer Communications and Networks, 2013
- S. Huang and J. Griffioen, "HyperNet Games: Leveraging SDN Networks to Improve Multiplayer Online Games", 18th International Conference on Computer Games, 2013
- S. Huang, J. Griffioen and K. Calvert, "Fast-tracking GENI Experiments using HyperNets", 2nd GENI Research and Educational Experience Workshop (GREE), 2013
- S. Huang J. Griffioen and K. Calvert, "PVNs: Making Virtual Network Infrastructure Usable", Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2012
- S. Huang, "Supporting Delay-intolerant Applications", Poster Proceedings of the 2008 ACM CoNEXT Conference, 2008

Shufeng Huang