January 2010

Abstractions and Algorithms for Control of Extensible and Heterogeneous Virtualized Network Infrastructures

Charles Wiseman
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Dissertation Examination Committee:
Jonathan Turner, Chair
Christopher Gill
Chenyang Lu
Keith Sawyer
Bill Smart
Tilman Wolf
Ken Wong

ABSTRACTIONS AND ALGORITHMS FOR CONTROL OF EXTENSIBLE AND HETEROGENEOUS VIRTUALIZED NETWORK INFRASTRUCTURES

by

Charles Gordon Wiseman

A dissertation presented to the
Graduate School of Arts and Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

August 2010
Saint Louis, Missouri

ABSTRACT OF THE DISSERTATION

Abstractions and Algorithms for Control of Extensible and Heterogeneous Virtualized Network Infrastructures

by

Charles Gordon Wiseman

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2010

Research Advisor: Professor Jonathan Turner

Virtualized network infrastructures are currently deployed in both research and commercial contexts. The complexity of the virtualization layer varies greatly in different deployments, ranging from cloud computing environments, to carrier Ethernet applications using stacked VLANs, to networking testbeds. In all of these cases, many users are sharing the resources of one provider and each user expects their resources to be isolated from all other users. There are many challenges associated with the control and management of these systems, including resource allocation and sharing, resource isolation, system security, and usability. Among the different types of virtualized infrastructures, network testbeds are of particular interest due to their widespread use in education and in the networking research community.

Networking researchers rely extensively on testbeds when evaluating new protocols and ideas. Indeed, a substantial percentage of top research papers include results gathered from testbeds. Network *emulation* testbeds in particular are often used

to conduct innovative research because they allow users to emulate diverse network topologies in a controlled environment. That is, researchers run experiments with a collection of resources that can be reconfigured to represent many different network scenarios. The user typically has control over most of the resources in their experiment which results in a high level of reproducibility. As such, these types of testbeds provide an excellent bridge between simulation and deployment of new ideas. Unfortunately, most testbeds suffer from a general lack of resource extensibility and diversity.

This dissertation extends the current state of the art by designing a new, more general testbed infrastructure that expands and enhances the capabilities of modern testbeds. This includes pertinent abstractions, software design, and related algorithms. The design has also been prototyped in the form of the Open Network Laboratory network testbed, which has been successfully used in educational and research pursuits. While the focus is on network testbeds, the results of this research will also be applicable to the broader class of virtualized system infrastructures.

Acknowledgments

I would like to thank my advisor, Dr. Jonathan Turner, for all of his guidance during my dissertation research. It has been a pleasure working with him, and I am particularly grateful for all the intellectual freedom he gave me to pursue topics that interested me most. His ability to understand theoretical subtleties while still keeping practical details in mind is incredible, and is something that I have benefited from greatly.

The students of the Applied Research Laboratory have all provided countless hours of discussion, both research related and not. It has been a pleasure working with them, and I hope that the friendships built in ARL will continue far into the future. My sincerest thanks also to all of the students that contributed to the Open Network Laboratory. I could not have accomplished all that I have without that help.

A special thanks goes to the staff members in the Applied Research Laboratory: Ken Wong, Jyoti Parwatikar, John DeHart, Dave Zar, and Fred Kuhns. Each of them has taught me an enormous amount ranging from technical details to what research is really about to how to work effectively in a team. Their advice, friendship, and knowledge has made ARL an unparalleled place to be a graduate student.

I would also like to thank the National Science Foundation for funding my research and providing me many opportunities to share my work with others.

There are countless other teachers, coworkers, and friends without whom I would not be where I am today. I want to thank two in particular who played a significant role in setting me on my current path: Guy Mauldin and L. G. Smith. I had the good fortune of taking classes from both of them at Science Hill High School. It was from them that I first began to understand the joy of teaching that has driven me to follow in their footsteps.

My family has continued to provide their loving support over the years of my education. They were always ready to encourage me when I needed it and never hesitated to offer anything they could do to help. I thank them also for patiently listening to all of my detailed rants and diatribes on obscure bits of computing technology.

Finally, I owe more than I can express here to my wife, Sandra. She kept me moving forward every day and helped me on many occasions to step back from the trees to see the forest again.

Charles Gordon Wiseman

Washington University in Saint Louis
August 2010

Dedicated to my wife.

Contents

List of Tables

List of Figures

List of Listings

List of Algorithms

Chapter 1

Introduction

Computer networking researchers today have an extraordinary opportunity to see their work affect people around the world via the Internet. Indeed, one of the strengths of the Internet is that anyone can develop new applications that anyone with an Internet connection can use. Networking researchers are generally more interested in fundamental questions about the structure of the Internet and the devices that make it up. This includes the design of new networking algorithms, protocols, and services that support new application classes. Of course, any such ideas must be rigorously tested and evaluated before being deployed in the actual Internet. Network testbeds provide an environment that is designed specifically to fill this role.

A network testbed is simply a collection of networking and systems resources that share a common infrastructure. The resources could be any network devices such as PCs, routers, and firewalls. The infrastructure that underlies the testbed resources takes many different forms depending on the design of the individual testbed. Commonly included are a unified authentication system for accessing the various resources, isolation mechanisms to ensure that concurrent experiments do not interfere with each other, and tools that aid in the configuration and use of the testbed resources.

Researchers have come to rely extensively on testbeds for developing, testing, and evaluating new ideas. Indeed, many papers published in top networking conferences present experimental results gathered in a network testbed. There are a number of reasons why this is true.

First, testbeds contain resources that are otherwise not available to many researchers. The largest testbeds have hundreds or thousands of nodes (mostly PCs). Smaller

research groups and universities do not have the capability to purchase and operate these large installations just for the purpose of conducting networking experiments. Some testbeds also include specialized or high-end equipment that is particularly difficult to acquire, mostly due to high cost. By placing such technology in a testbed environment, many more researchers and educators benefit for each dollar spent.

Second, testbeds allow new ideas to be evaluated across a wide range of network topologies and scenarios. Many testbeds give users the ability to reconfigure experimental network topologies explicitly. Others provide resources at a large variety of network locations, which leads to diverse network paths between the set of communicating pairs. In either case, the result is that users can experiment with network conditions that would be very difficult to reproduce in a local, non-shared lab setting.

Third, many testbeds open avenues of research that are difficult or impossible to pursue in the commodity Internet. The extent to which this is true varies greatly from testbed to testbed. Some testbeds support network services that are simply not available in the current Internet, such as multicast or network path provisioning. Other testbeds allow users to replace various layers of the standard networking protocols with enhanced or entirely new protocols, e.g. to test a new congestion control algorithm.

Fourth, testbeds provide a common platform for comparing similar research projects. Of course, this is dependent on researchers publishing their methodology and results, but when they do, it allows other groups to more easily make fair comparisons to their work. This also encourages researchers to report their own results accurately and fairly.

Network testbeds also provide an excellent avenue for educators and students to explore practical aspects of computer networking and systems. Indeed, all of the above reasoning also applies in an educational context. Testbed environments give students the opportunity to get hands-on experience across a variety of networking devices and/or scenarios without the risk of damaging a production system.

For all of these reasons, network testbeds are an important class of systems that have become essential to enabling ongoing research in networking. Moreover, there are challenging research opportunities that arise during the design of testbeds, including

resource allocation, resource isolation, heterogeneity, extensibility, ease of use, and ease of management. Our experience building a local testbed has exposed many of these challenges. The National Science Foundation's Global Environment for Network Innovations (GENI) [26] project, which aims to build a generalized large-scale testbed, has further heightened community interest in how these types of systems should be designed.

This dissertation explores the current state of the art in networking testbeds in order to design and build a more sophisticated testbed environment. To begin, an overview of related work is given next.

1.1 Related Work Overview

There are a few different areas of important related work to discuss. We will start with a description of some of the more popular existing testbeds. There are two broad categories of network testbeds: *overlay* and *emulation*. Overlay testbeds exist as overlay networks on the Internet and as such allow testing under the actual conditions present in the Internet. They also provide the ability to deploy long-running services on the Internet, although isolation among concurrent users is typically very poor. On the other hand, emulation testbeds allow users to emulate different network topologies and conditions in a controlled and strongly isolated environment. This is normally accomplished by separating the testbed environment from the Internet and connecting the testbed resources with switches and routers that support many virtual networks on top of the same physical substrate.

The most widely used network testbed in current research is PlanetLab [55], which is an overlay testbed that has been in operation for over five years. PlanetLab consists of a large number of PCs with Internet connections scattered around the globe. At the time of this writing, there are over 1000 PlanetLab nodes at over 475 sites. Each PlanetLab node can support multiple concurrent experiments by instantiating one virtual machine on the node for each active experiment. The PlanetLab control software takes user experiment requests for a set of nodes and contacts the individual nodes to add the virtual machines for the user. Researchers use PlanetLab to debug and refine their new services and applications in the Internet context. They

can also advertise services and applications, and allow others (on PlanetLab nodes or not) to treat them as an actual deployment. PlanetLab's success has resulted in large numbers of active experiments, particularly before major conference deadlines. Unfortunately, PlanetLab has no admission control and no limit on the number of experiments on each node. This leads to extremely poor and unpredictable performance as each active experiment is competing for both processor cycles and the limited network bandwidth available on the node.

There have been a few efforts to overcome the deficiencies of PlanetLab through the design of new nodes that enable better and more consistent performance while still operating in the PlanetLab context. The Supercharged PlanetLab Platform [66] and VINI [9] are both in their early stages of deployment and have the potential to enable a broader set of useful experiments than standard PlanetLab, but neither project will result in substantial changes to the PlanetLab control and management scheme. Nevertheless, the design of the nodes themselves will be useful when we consider what abstractions are needed to represent arbitrary network resources.

SatelliteLab [24] is a newer overlay testbed that supports heterogeneous edge devices. That is, the testbed contains desktop PCs, laptop PCs, and handheld devices that are connected to the Internet over a range of link types, including broadband, ISDN, Wi-Fi, and cellular connections. Users can then test their applications across widely varied network conditions. Of course, it would be difficult to have user-written code running on devices like cell phones, so they break their architecture into two tiers. Standard PlanetLab nodes form the backbone of the testbed and that is where users run their applications. The edge devices only forward traffic with pre-built code from the testbed operators. As a result, SatelliteLab provides users with network heterogeneity that is lacking in other testbeds, but does not actually allow users to experiment with code on anything other than PlanetLab PCs.

Another overlay testbed is the Resilient Overlay Network (RON) testbed [4]. RON is similar to PlanetLab in the sense that it is a collection of PCs with Internet connections, but it is a much smaller-scale testbed with less than 40 nodes. The limited size of the testbed led RON to be managed in a much different way than PlanetLab. Namely, it is a cooperative environment where users are given logins on every node in the testbed and are then expected to coordinate usage out-of-band. The testbed

operators reserve the right to revoke or deny any accounts, but even that happens strictly through manual intervention. While there is definitely appeal to an open, friendly system like this, it clearly does not scale to large systems with many users.

Moving on to emulation testbeds, Emulab [69] is another one of the most popular testbeds. Once again, this testbed provides access to a large number of PCs. Emulab nodes have at least two network interfaces. One interface is used solely for the control network so that users have out-of-band access to the nodes for the purposes of configuration and monitoring. The other interface(s) can be configured according to the needs of each individual experiment so that the nodes can act as end hosts or routers. As with other emulation testbeds, Emulab uses a small number of switches and routers as a programmable patch panel that indirectly connects all of the PCs. Virtual Local Area Networks (VLANs) are used by the switches and routers to allow arbitrary topologies to be built on top of the PCs while also ensuring that traffic in each experiment is isolated from traffic in others. Additional nodes are automatically inserted behind the scenes so that users can configure link properties in order to emulate network contexts that do not exist in a fixed LAN.

The Emulab software has been made available so that other groups can use it to run their own testbeds. A number of these Emulab-powered testbeds are currently operating, although most of them are not open to the public. One exception is the DETER testbed [11] that is focused on security research. The control infrastructure is identical to Emulab, but additional software components were added to ensure that users researching security holes and other dangerous exploits remain both contained in the testbed and isolated from other experiments. The Wisconsin Advanced Internet Laboratory (WAIL) [67] is another such testbed that utilizes the Emulab software base. WAIL is unique among the Emulab testbeds in that they have extended the software to support commercial routers as fundamental resources available to researchers. The documentation on the website indicates that users do not have write access to router configurations by default, and, unfortunately, there is not yet any detailed documentation that discusses how they have modified the Emulab model to support these heterogeneous resources.

The Open Network Laboratory (ONL) is another emulation testbed that is similar in some respects to Emulab. ONL was originally built around a small number of

hardware routers with a focus on performance evaluation studies. In recent years, ONL has been expanded to include more diverse types of networking technology and has now been used as a prototyping environment for the ideas and algorithms developed as part of this dissertation. A more detailed description of ONL is provided in Chapter 2.

There is another class of testbed that is not being considered directly in our work, but that faces some of the same issues as emulation and overlay testbeds: wireless sensor network testbeds. Numerous wireless sensor network testbeds have been deployed and studied, and an overview of the most prominent such testbeds is given here. Wireless sensor network testbeds are usually used only locally by the research group that deployed them. Motelab [68] and Orbit [57] are two exceptions. In each case, users can access the testbed remotely over the Internet and upload their own code on to the sensor motes. The nature of sensor networks led to a simple and effective control infrastructure that allocates the entire sensor network to one user at a time. Users make reservations ahead of time, and the control software ensures that only the user who has the current reservation has access to the motes. Emulab also has an extension for wireless sensor networking called Mobile Emulab [37] that leverages the existing Emulab control software. Finally, projects such as TWIST [34] and Sensenet [23] are not aimed at sharable testbeds, but do describe fairly generic infrastructures that could be used by others who wish to deploy their own wireless sensor network testbeds.

Simulations are also used extensively in networking research for initial exploration and evaluation of new algorithms and protocols. Researchers often build their own simulators that are tailored to their current needs. There are a few more commonly used simulation platforms, such as ns-2 [50], which is a discrete event simulator that has native support for common networking protocols and components. In general, simulations provide an excellent first step towards realization of any new idea. They are, however, limited in scope due to the difficulty in building a high fidelity simulation of an entire network. It requires a substantial effort to faithfully simulate every aspect of an operating network. Moreover, the time to run such simulations can become prohibitive. Some efforts have been made to build distributed network simulators [40, 60], but it is usually easier (and faster) to move to real networks once simulations

become too large. Emulation testbeds in particular ease the transition away from simulation by providing controlled and contained environments for experimentation.

There are a few other areas of networking research that are related to testbeds. Cloud computing has recently become a popular topic of discussion both in the public and among researchers. The actual definition of cloud computing has been much debated [7], but cloud computing infrastructures share many properties of testbed infrastructures. Indeed, in both cases, users run their own applications on resources located somewhere in the Internet that are controlled by a single entity. As such, the design of cloud computing systems is of interest here. A few research proposals exist that address this directly, e.g., Eucalyptus [51] and Seattle [12]. There are also a number of commercial cloud computing solutions such as Amazon's EC2 [2] and Google's AppEngine [28]. Although little has been published about the design internals of these commercial clouds, it is still possible to use them as examples for usage patterns when comparing cloud computing to network testbeds.

Finally, there are some common elements in the control and management of testbeds and of general operational networks. In each case, there are potentially large numbers of distributed resources that have to be managed by the provider. The actual research questions for the two areas do not overlap much, but the mechanisms used to address network control plane issues are similar. For example, testbeds typically support end-user-driven experiments that utilize some set of the testbed resources, while an operational network is configured by the network provider in response to a set of high-level policy decisions. For example, Internet Service Providers use VLAN stacking to provide Carrier Ethernet services to large enterprise customers in order to allow physically separate locations to appear logically co-located. The ISP must manage VLANs and network capacity across all such customers. Both cases also rely on easy deployment of changes to the underlying networks. This, in turn, requires a software infrastructure that can present all the relevant information to the user/operator and then enact changes across the entire network. Most network management solutions are tied to the actual network equipment such as Cisco's IOS [16]. There are other commercial solutions like OpenNMS [52] that are less hardware-dependent but generally do not scale well. Some research has also been done to try to simplify the entire management plane by redesigning the control protocols from the ground up, e.g., the 4D architecture [29].

1.2 Summary of Chapters

The rest of the dissertation is organized as follows.

Chapter 2 describes a new, more general testbed framework that encompasses important features of current testbeds and adds new features such as resource heterogeneity and extensibility. Design goals and choices are given for this new framework, followed by the set of core abstractions and software components that comprise the framework. One specific instance of a testbed based on this framework, the new Open Network Laboratory, is described in detail including implementation details and usage examples.

Chapter 3 provides an overview of a set of packet processing platforms that are available in the new Open Network Laboratory and have been specialized for use in the testbed environment. The Network Processor-based Router, which was developed as part of this dissertation research, is described in detail.

Chapter 4 discusses the problem of scheduling and reserving resources in a testbed environment. A general framework for solving this type of scheduling problem is described, and results are given that highlight the scheduler's performance in the Open Network Laboratory.

Finally, Chapter 5 concludes the dissertation.

Chapter 2

Testbed Infrastructure

Virtualized network and system infrastructures are becoming more and more commonplace across a variety of commercial and research deployment contexts. Cloud computing environments use end system virtualization to allow many applications to share computational and storage resources owned by one cloud provider. ISPs use VLAN stacking to provide carrier Ethernet services across wide area networks to large enterprise customers. Network testbeds use various types of virtualization to share network nodes and links among many concurrent virtual network experiments running on the shared testbed substrate. Fundamentally, these types of systems use virtualization to share the resources of one provider among many customers or users. More importantly, the resources assigned to each customer are isolated from those of other customers.

Configuration and management in non-virtualized infrastructures is already a difficult task. Supporting virtualization layers adds further complexity whereby providers must configure their networks to meet the isolation and performance requirements of all of their customers. Our work in this area is focused on network testbeds, although the ideas explored here could be applied in other virtualized infrastructure settings.

Researchers have come to rely extensively on testbeds for developing, testing, and evaluating new ideas. Interestingly, nearly all existing testbeds have only PCs as user configurable resources. There are clear benefits to this choice, including simpler testbed management software and user familiarity with the PC platform. However, a testbed with a diverse collection of heterogeneous resources would provide a number of benefits to the research community. Researchers would be able to conduct

experiments with a variety of networking devices and technologies in a contained environment. This is also useful for educators who are interested in giving students hands-on experience with networking technologies other than PCs. Natural additions to such a testbed would include network processor systems, FPGAs, and other hardware-based or commercial devices. Access to these types of high performance, reconfigurable nodes would also allow researchers to test new protocols and ideas under realistic conditions while still operating in an isolated setting. PCs alone can certainly be used to emulate the functionality of many different devices, but they can not sustain Internet-scale throughput or match delay characteristics of more specialized technologies.

Most existing testbeds also provide only a low level interface to users for configuring nodes in their virtual networks. This is a reasonable choice for PC-only testbeds, although including higher level tools would reduce the manual configuration burden. In the case of a testbed with heterogeneous components, it is necessary to include configuration interfaces that export higher level abstractions to users (e.g., routes, packet filters, queueing parameters, etc) as most users will not be familiar with the native configuration tools for resources such as FPGAs or network processors. Of course, many types of devices can be reprogrammed to support a range of networking functionality. It is thus important to support extensible resources, where users can select among the different possibilities for each device type and even add and configure their own new functionality if necessary.

Based on the above observations, this chapter describes the design, implementation, and usage of a new, more general testbed framework that encompasses and extends the current state of the art. The framework is targeted at testbeds that have provisioned interconnections (emulation or overlay), although it is applicable with some modifications to any testbed. For simplicity, it will be described it the context of an emulation testbed.

2.1 Design

Some terminology is now defined before the actual design is given. A *node* is one piece of hardware in the testbed that a user might request. This could include PCs,

switches, routers, etc. A *session* is one instance of a virtual network running on the testbed. Each session begins when the user requests resources from the testbed and ends when those resources are released back to the testbed. Sessions generally range in length from a few minutes to a few days, although they could run for longer periods.

Most emulation testbeds share a few basic design objectives. The first is the ability to support multiple concurrent sessions in the testbed. This clearly leads to better utilization of the testbed resources as the number of nodes in the testbed grows.

Objective: The testbed can host many concurrent sessions.

It is also common to provide isolation among sessions, i.e., actions in one session should have no impact on other concurrent sessions. Users are thus given the impression that their session is running in a native, unshared environment rather than a testbed. This is also necessary for a high level of reproducibility, which is one of the major benefits of running sessions in an emulated environment.

Objective: Concurrent sessions are isolated from each other.

One of the obvious weaknesses of most existing testbeds is a lack of node heterogeneity. Indeed, only a very small number of testbeds offer nodes other than PCs. Those that do, such as WAIL and SatelliteLab, do not actually let users configure those nodes in any substantial way. A testbed that contains a variety of nodes would be a significant boon to the research community. Researchers (and educators) would be able to conduct sessions with a variety of networking technologies in a safe, controlled environment. Access to high performance, configurable devices would also allow researchers to test their protocols under realistic network conditions while still in a testbed context. PCs can certainly be used to emulate the functionality of many different network devices, but they are not capable of sustaining Internet-scale throughput or matching delay characteristics of specialized technology. As such, we feel that building an emulation testbed framework that supports diverse networking

technologies would be an important step forward. We will use *resource type*, or simply *type*, to refer to a node class. That is, each node in the testbed is of a specific type, such as an Ethernet switch, a programmable router, or a PC.

> **Objective:** The testbed must support a wide range of diverse networking devices.

It is also important for testbed control and management to remain as simple as possible. This is particularly important when it comes to adding new resource types to the testbed. The testbed infrastructure should be designed to minimize the effort of adding new types, and ideally it should be handled without the need to modify any of the testbed software. This burden is entirely on the testbed managers, not the users, but reducing the time spent on ongoing management tasks frees up testbed staff to improve the testbed in more substantive ways. In other words, the testbed itself should also be easily extensible.

> **Objective:** The testbed infrastructure must be extensible to support simple management.

One of the primary reasons for having a testbed that supports heterogeneous resources is for users to interact and gain experience with varied networking technologies. It is difficult for that to happen if there is very little visibility into the operation and configuration of the nodes. Of course, not every resource type is user-configurable. For example, simple learning Ethernet switches do not require any configuration. Most resource types, however, have many control knobs and configuration possibilities. Users will benefit the most when they have access to as many of those options as possible.

> **Objective:** The testbed must support exposure of complex configuration interfaces.

On the other hand, too many configuration options can make it difficult for novice users to get started. This problem is only compounded if there are many different resources types each of which has its own set of control knobs to learn. One solution is to keep the resources opaque and not give users too much control. Since that violates the previous design objectives, a better approach is to enable the testbed software to make session configuration and execution simple for new users while still providing expert users a higher level of node control.

> **Objective:** Configuring sessions must be simple enough for novices.

A related consideration is how resources are shared among users. One approach is to virtualize all the nodes in the testbed and allow multiple users to share the same node simultaneously. This is clearly not a viable option here, as many networking devices do not support virtualization. PCs are one of the few types that are easily virtualized, although PC virtualization typically leads to poor performance isolation. Another approach is to allocate nodes solely to one user at a time, which fits more closely with the previous objective. Of course, the testbed may only have a small number of nodes of some resource types, so a reservation-based scheduling system should be used to allow users to share nodes easily over time. Scheduling sessions in this context is not a trivial problem, but it does provide an effective means to share testbed resources while still meeting the other design objectives. One possible scheduling framework is described in detail in Chapter 4. Note that virtualization is still used in the testbed substrate network in order to provide traffic and link isolation (e.g., by using VLANs on backbone switches to separate traffic on different virtual links). The testbed as a whole is, of course, a virtualized platform, whether or not individual nodes in the testbed are virtualized.

> **Decision:** Nodes are only assigned to one user at any given time and are shared according to some resource scheduling policy.

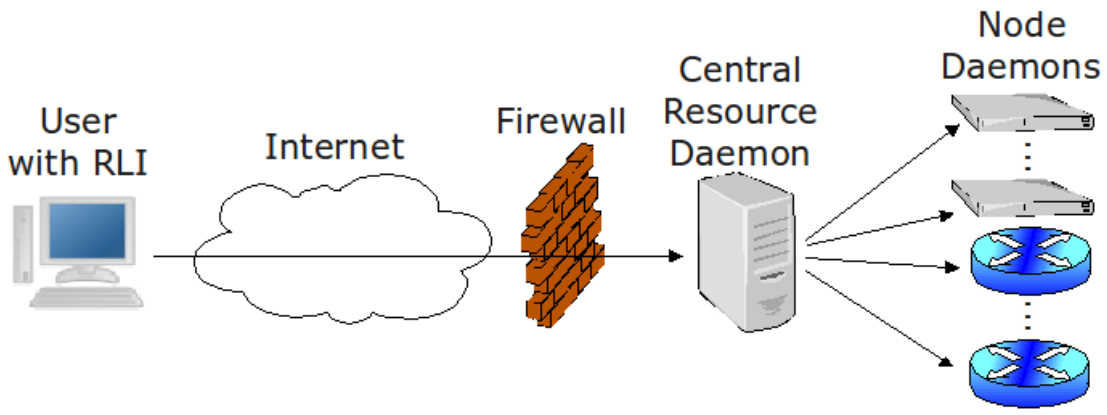

Figure 2.1: Testbed software overview.

2.2 Framework

One possible testbed framework is now presented that is based on the design choices in the previous section. The software infrastructure is given first followed by the core abstractions that allow the infrastructure to meet to the design objectives.

2.2.1 Infrastructure

An overview of a general software infrastructure is shown in Figure 2.1. There are three main pieces: the Remote Laboratory Interface, the Central Resource Daemon, and the Node Daemons. There is also a testbed firewall whose primary function is to keep experimental traffic from leaving the confines of the testbed network, although it also serves as a normal firewall that blocks external malicious traffic.

The Remote Laboratory Interface (RLI) is the user interface to the testbed. It runs on the user's computer and is used to build session topologies, configure nodes, and monitor the user's experimental network. Clearly, the RLI must present a consistent and intuitive view of the diverse resources in the testbed. The primary goal of the RLI is to make user interaction with testbed resources as easy as possible for novice users without restricted the needs of more advanced users.

The RLI communicates with the Central Resource Daemon (CRD) on behalf of the user. The CRD is responsible for instantiating and removing sessions. Whatever

scheduling policy is used for the testbed would be invoked by the CRD to ensure that the user has the right to use all of the needed resources for their session at that time. It also ensures that every node in the user's topology is in the proper state before giving them access to the node. Finally, the CRD relays control messages from the RLI to the nodes rather than having the RLI contact nodes directly. This way, the RLI only needs a single point of contact for the testbed, keeping firewalls (testbed and user) simpler. Note that in practice, the CRD may be split into many coordinating parts that fill the different roles played by the CRD.

The Node Daemons (NDs) accept control messages from the CRD and the RLI. Not every resource type will have a Node Daemon (e.g., a simple Ethernet switch), but most will. There is one ND for each resource type that supports it, and every node of that type runs the daemon. The ND understands how to interact with the node and makes any necessary changes based on the control messages received.

2.2.2 Abstractions

Given the above simple software structure, this section will fill in the details of how the various components actually work together to support the design objectives for the testbed. The most important abstraction to consider is the resource type abstraction. Designing this abstraction correctly is critical to allowing the testbed infrastructure to support resource heterogeneity and extensibility. This in turn allows the testbed to include highly configurable and programmable resources that can be utilized in a variety of ways by the users.

Each type is represented in two parts: the *base* description and the *specialization* description. The base description is a representation of the physical device, e.g., a PC, network processor card, or FPGA. Although this description could encompass many things about the hardware, there is no need for the core of the description to be complex. Indeed, the base description of each type only needs to contain a unique identifier and a list of network interfaces. That is enough to accurately represent a node in a virtual network topology in the RLI and in the CRD. Figure 2.2 shows a possible XML [18] example for a network processor card that has four 1 Gb/s Ethernet ports.

```
<base-type id="np-card">
  <network-interfaces>
    <port number="0" linktype="GigE" />
    <port number="1" linktype="GigE" />
    <port number="2" linktype="GigE" />
    <port number="3" linktype="GigE" />
  </network-interfaces>
</base-type>
```

Figure 2.2: Base description for a 4 port network processor card.

When the user starts a session, the RLI sends the virtual network topology to the CRD. The CRD works within the testbed resource allocation policy to assign instances of the required types to the user and to configure the testbed substrate for the session. All assignment decisions and operations in the CRD use only the base descriptions of each node in a user's topology.

The second part of the type representation is the specialization description. The specialization description framework allows the RLI to present a simple user interface to the user that encompasses many different resource types. The specialization abstraction represents one set of functionality supported by a particular type. For example, an IPv4 router built on top of a network processor card, or a traffic generator built on top of a NetFPGA. Ideally, the RLI will use the specialization description to automatically provide all of the GUI menus, tables, etc. for the type. This would allow new types to be added to the testbed without the need to change the RLI. Of course, the description specification needs to be fixed so that the RLI can parse it, but general enough to support most common user interactions with diverse networking technologies.

There are two basic ways for users to interact with a node. They can send configuration updates to the node, and they can monitor the node. Configuration updates can include anything that changes the operational state of the device. For example, a router might accept configuration requests to modify the routing state and to configure queues. In general, networking devices keep most of their operational state in tables or data structures that can be viewed as tables. The specialization description will thus include generic table elements that can be tailored for each device as part of the specialization. Every table and general configuration command is either

associated with the global node state or per-port state. For example, a router may have a separate routing table at each input port, or a global routing table shared by all input ports. Monitoring requests are used to get real-time data from the device. Continuing the router example, it may support monitoring the packet rates at each network interface as well as monitoring current queue lengths. Monitoring commands are also tied either to a particular port or to the node globally.

These two classes of interaction provide a generic framework for specifying the interface exported by a device. Configuration updates consist of a message type and a series of parameters to accompany the request. If the update is associated with per-port state on the node, the port number is also sent with the request. The RLI automatically adds a GUI menu item for that type that gathers the needed parameters and then sends the entire request to the ND for the appropriate node. In the case of a table, the RLI generates add, remove, and update entry messages for the user as they modify the table in the RLI. The ND calls a message handler for the request that understands how to parse the parameters and enact the update on the node. This restricts the specialization-specific software to the NDs. The ND sends any result information back to the RLI, which is often a simple success or failure code, but could also include a series of parameters that will be displayed to the user.

Monitoring requests are represented the same way, as a message type and a series of parameters. The ND again calls a message handler to respond to the request. In this case, the response will consist of a timestamp and the requested data. Requests can also be periodic, e.g., monitoring a value every second and returning the result. The RLI uses the results to display real-time charts of the data that allow the user to track soft state in their virtual network.

Figure 2.3 shows one possible specialization description for the example router described above. This specialization has one configuration table, one configuration command, and three monitoring commands. The table contains routing information, where each table entry consists of the standard CIDR address range for this route, and the output port and next hop IP addresses for matching packets. The other configuration command is for setting the queue threshold (number of bytes the queue can store) and quantum (a parameter of the packet scheduling policy) for one particular queue. The monitoring commands are, in order, for monitoring the number

```
<specialization id="ipv4-router" base="np-card">
  <configuration>
    <table name="route table">
      <param name="cidr-address" type="string" />
      <param name="output-port" type="integer" />
      <param name="next-hop-ip" type="string" />
    </table>
    <command name="set-queue-param">
      <param name="queue" type="integer" />
      <param name="threshold" type="integer" />
      <param name="quantum" type="integer" />
    </command>
  </configuration>
  <monitor>
    <command name="rx-packet-count">
      <param name="port" type="integer" />
    </command>
    <command name="tx-packet-count">
      <param name="port" type="integer" />
    </command>
    <command name="queue-length">
      <param name="queue" type="integer" />
    </command>
  </monitor>
</specialization>
```

Figure 2.3: Simple specialization description for an IPv4 router.

of packets received on one particular port, the number of packets transmitted out one particular port, and the current length (number of bytes currently stored) in one particular queue.

Together, one base and one specialization description represent a resource type. Each base may have multiple specialization types associated with it, but each specialization type has only one base. This separation allows hardware resources to support many different types of functionality. This is particularly useful for technologies like network processors or FPGA platforms that can be programmed to function as a variety of network devices. It also provides a means to support multiple interfaces for a type that are targeted at different levels of expertise. A novice interface might only support the most basic interaction with a device while an expert interface would expose every possible control knob.

2.2.3 Implementation Details

The RLI parses all available type and specialization descriptions to build menus for adding each type and each specialization to a user's virtual network. Note that users can add types without any specialization if they do not need a higher level interface and plan to configure every node manually. Once added to a topology, each specialized node can be configured according to its specialization description via menus and dialogs that are accessed by clicking on the node.

By design, the RLI is not part of the trusted code base for the testbed because it is run on remote PCs under the control of users. The CRD is therefore responsible for all security checking, user authentication, and messaging protocol verification.

Once the user is ready to start a session, the RLI sends the user's virtual topology to the CRD to instantiate the session. The CRD only uses the type description, not the specialization description, for each node. As mentioned above, the CRD has four primary functions. To keep the implementation clean, some of those functions are broken out into separate software processes as shown in Figure 2.4.

The Relay accepts messages from all user RLIs and routes them either to the Node Daemons or to the Session Manager as needed. This serves as the gateway to the testbed for the RLIs. It also performs simple security and isolation checks to ensure that users attempting to send messages to a particular node actually have that node assigned to them in a current session.

The Session Manager handles the bulk of the work in the CRD and maintains all state related to the testbed. This state includes current session information, reservations, and descriptions of the nodes and physical network for the testbed. Adding new types or new nodes to the testbed is accomplished by adding new entries into tables in the testbed database and requires no modifications to the Session Manager or any other ONL software components. The scheduler subsystem is invoked to add and remove reservations. It is also called to verify that the user has a valid reservation when a new session request arrives. The Session Manager then initiates the new session by performing any initial configuration for all the nodes and links in the user's virtual network.

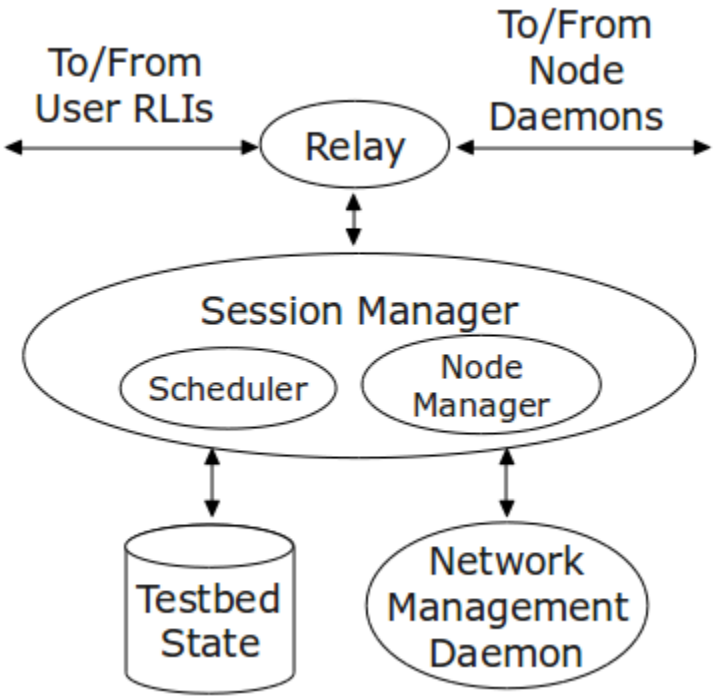

Figure 2.4: Software Components of the Central Resource Daemon.

Configuring the underlying physical testbed network is handled by the Network Management Daemon. The Network Management Daemon supports two basic functions: adding/removing a virtual switch and adding/removing nodes to/from a virtual switch. The actual mapping from virtual links and switches in a user virtual network to the physical links in the testbed network is handled by the scheduler in the Session Manager. The Network Management Daemon simply adds or removes the mappings as requested. In general, any virtualization technology supported by the underlying network could be used by the Network Management Daemon. In practice, the most common approach would be to configure and remove VLANs for each switch or link in the user's virtual network. This functionality is not complex, but it could be tied closely to the types of switches and routers used in the testbed. Separating it from the Session Manager allows this infrastructure to be more easily adapted to changes in the physical network as only the Network Management Daemon would need modification.

The Node Manager subsystem of the Session Manager contacts the appropriate Node Daemons for any initial configuration. The Node Daemons are split into two parts,

corresponding to the base type and specialization of the node. The *Base* Node Daemon is responsible for initializing the node at the beginning of a session and for bringing the node back to a known state at the end of a session. It also starts the appropriate *Specialization* Node Daemon at the end of its initialization. The Specialization Node Daemon interacts with the RLI to respond to configuration and monitoring requests defined in the specialization description.

This split enforces a clean separation between the specializations for every type. More importantly, it allows users to add their own specializations for a type by providing the specialization interface description and the Specialization Node Daemon associated with it. For example, a user might add a firewall specialization to a PC by writing a simple Specialization Node Daemon and description for adding and removing firewall filtering rules. Note that the Specialization Node Daemon is a standard socket program and so users will be free to use any programming language to build one. None of the testbed software infrastructure needs to be modified to handle the new specialization.

2.3 Aggregate Types

A recent trend in networking research is to build high performance systems out of many distinct components [5,62]. For example, one might build a 10 Gb/s router out of 10 PCs each with a 1 Gb/s NIC connected to a common Ethernet switch. This *aggregate* node would then be configured and operated as if it were one single piece of equipment. The framework described above is also very well suited to support this type of research.

In the testbed framework as described above, users could add nodes and links individually to their session and logically view that collection as a aggregate node. Of course, configuration would then have to be done manually to each of the nodes in the aggregate, rather than configuring the single aggregate node directly. Instead, the framework includes an additional abstraction specifically to facilitate usage of these aggregate nodes.

The *aggregate* description provides the RLI with everything needed to treat an arbitrary collection of resources as a single entity. There are three main sections to each aggregate description file: the resource set, the device description, and the interface description. An example is shown in Figure 2.5 for a 5 port router based on 5 PCs that are specialized as standard end hosts.

The resource set defines all node and links that make up one instance of an aggregate node. Each node in the aggregate can be either a base type or a specialization. The links specify how all the nodes in the aggregate are interconnected. Together, they describe exactly what the testbed infrastructure will need to allocate for each instance of the aggregate type in question.

The device description defines how the aggregate node will actually look to the user. Primarily, this means a listing of the logical network interfaces that the user will interact with. In this regard, the device description is very similar to the base type description for normal nodes.

Finally, the interface description contains all of the configuration and monitoring commands supported by the aggregate node. This is precisely the same as the specialization description for non-aggregate types.

In Figure 2.5, the resource set consists of 5 PCs that are running the "end-host" specialization, and an Ethernet switch. Each node in the set is given a label that is used in link specifications to dictate how the nodes are connected to one another. For example, the first link shows that "pc1" port 0 is connected to "switch" port 0, where "pc1" refers to the first node in node list, and "switch" refers to the last node in the nodes list. The device description is similar to the base description, as in Figure 2.2, and in this case indicates that this aggregate type should be displayed as a single device having 5 ports. Each port listing also describes how that aggregate port relates to a physical port on one of the nodes within the aggregate. That is, when a link is attached to one of the aggregate ports it must actually be attached to one port on one of the nodes in the aggregate. In the example, each of the 5 ports is connected to a different port on the "switch" node. Lastly, the interface description uses the same structure as Figure 2.3, with configuration and monitoring commands that are appropriate to an IP router.

```
<aggregate id="pc-router">
  <resources>
    <node label="pc1" base="pc" specialization="end-host" />
    <node label="pc2" base="pc" specialization="end-host" />
    <node label="pc3" base="pc" specialization="end-host" />
    <node label="pc4" base="pc" specialization="end-host" />
    <node label="pc5" base="pc" specialization="end-host" />
    <node label="switch" base="ethernet-switch" />
    <link end1="pc1" end1port="0" end2="switch" end2port="0" />
    <link end1="pc2" end1port="0" end2="switch" end2port="1" />
    <link end1="pc3" end1port="0" end2="switch" end2port="2" />
    <link end1="pc4" end1port="0" end2="switch" end2port="3" />
    <link end1="pc5" end1port="0" end2="switch" end2port="4" />
  </resources>
  <device>
    <network-interfaces>
      <port number="0" node="switch" nodeport="5" />
      <port number="1" node="switch" nodeport="6" />
      <port number="2" node="switch" nodeport="7" />
      <port number="3" node="switch" nodeport="8" />
      <port number="4" node="switch" nodeport="9" />
    </network-interfaces>
  </device>
  <interface>
    <configuration>
      <table name="route table">
        <param name="cidr-address" type="string" />
        <param name="output-port" type="integer" />
        <param name="next-hop-ip" type="string" />
      </table>
    </configuration>
    <monitor>
      <command name="rx-packet-count">
        <param name="port" type="integer" />
      </command>
      <command name="tx-packet-count">
        <param name="port" type="integer" />
      </command>
    </monitor>
  </interface>
</aggregate>
```

Figure 2.5: An example aggregate description for a 5 port router built with 5 PCs and a switch.

As with the base and specialization descriptions, the aggregate description has an accompanying *Aggregate* Node Daemon. Every instance of an aggregate node results in a separate instance of the Aggregate Node Daemon being started. The Aggregate Node Daemon is responsible for receiving and processing the messages for the aggregate node, as with the Specialization Node Daemon. In this case, however, the Aggregate Node Daemon acts primarily as a request translator and relay. It takes configuration and monitoring requests in the context of the aggregate node and then formulates and sends new requests to the individual Base and/or Specialization Node Daemons that are part of the aggregate. One original request to the Aggregate Node Daemon could cause zero or more new requests to be sent to other Node Daemons. The Aggregate Node Daemon then gathers the responses (if any) and sends its response back to the RLI.

As an example, consider again the aggregate description in Figure 2.5. The "pc-router" aggregate type has a global routing table. When an entry is added, the RLI will send one message to the Aggregate Node Daemon with the request to add a route. The Aggregate Node Daemon might, in turn, send a request to each of the five PCs in its resource set to add a local route on that PC. After getting the responses back from all five Specialization Nodes Daemons, the Aggregate Node Daemon would send a success or failure reply back to the RLI. On the other hand, when the RLI sends a request to monitor the received packet count on port 0 of the aggregate node, the Aggregate Node Daemon might send only a single request to the PC that is acting as port 0.

Treating aggregate nodes in this way results in many of the same benefits to the testbed infrastructure and users as with specializations. Most importantly, it allows users to configure aggregates the same way that they do any other node. The RLI is the only software component that needs to be aware of aggregates. It uses the description to present a single node to the user, but the session information it sends to the CRD contains the actual base node topology representation of each aggregate node. The CRD has no knowledge of aggregates. The Node Daemons likewise have no need to change based on whether or not the node is part of an aggregate. They respond to the requests they receive as normal, from the RLI or from an Aggregate Node Daemon.

The only caveat is that the Aggregate Node Daemon for each aggregate has to be started by some other daemon in the testbed. There are several ways to accomplish this. Perhaps the simplest is to allow the Base Node Daemons to start Aggregate Node Daemons, and have the RLI elect one of the nodes in the each aggregate to act as the Aggregate Node Daemon.

The result is that new aggregate types can be added at any time by any user without the need for any of the existing software to change. The new aggregate description must be provided and the associated Aggregate Node Daemon written, but nothing else in the infrastructure is aware of the new aggregate type. Moreover, this framework allows aggregates of aggregates naturally and without any additional complications.

2.4 Resource Scheduling

As discussed in Section 2.1, testbeds must be able to support many concurrent sessions that share the available resources in the testbed. This work focuses on testbeds that assign non-virtualized, non-shared nodes to every node in each user's virtual topology. This assignment is done with a testbed *scheduler*.

Recall that each physical node is given to at most one user at any time. The scheduling policy is thus used to determine how nodes should be shared when the demand is higher than the capacity for any particular resource type. The full description of the problem and one possible solution are given in Chapter 4, but a brief overview of the problem is given here.

There are two basic scheduling approaches: resources are either given on-demand or reserved in advance. In the former case, the scheduler looks strictly at the physical resources that are not in use by any other current session, as in standard admission control. This is how most emulation testbeds operate. The latter case is somewhat more complicated. The scheduler must keep a time line of *reservations* that determines which resources are available at any given time. In addition to the virtual network to be emulated, user requests include a period of time when that virtual network should be active. When a new request is made, all previously accepted reservations with overlapping times are considered. Any reservations whose start time

has not come (i.e., are not yet active) could potentially be remapped to a new set of physical resources, if necessary to "make room" for the new reservation. Clearly, maintaining a schedule of network mappings is a generalization of pure on-demand admission control. This dissertation will consider this more general problem.

Note that scheduling virtual networks in testbed environments has been studied previously [3, 58, 59]. It is a variant of the general network embedding problem, long known to be NP-hard. As such, heuristic approaches will be considered.

The scheduler takes a virtual network request and attempts to find a mapping from the virtual network onto the available physical resources. If a mapping is found, the physical resources in that mapping are added to a reservation for the user. The scheduler ensures that each physical node is mapped to no more than one virtual node across all virtual networks. It also ensures that there is enough capacity in the underlying testbed network to support every virtual link without causing interference with other virtual links (within the same reservation or not). Of course, the scheduler has to consider every previously accepted reservation when finding a new mapping, and any mapping that meets these requirements could be returned by the scheduler. Figure 2.6 shows a simple example mapping. In the figure, different shapes represent different types, with rectangles representing the infrastructure switches that are hidden from the user. The edge labels are edge capacities. Node labels show an example mapping from nodes in the user network to nodes in the testbed network. The dashed lines show one mapping from an edge in the user network to the corresponding path in the testbed network.

2.5 The Open Network Laboratory

The Open Network Laboratory has been operating for a few years now [22]. It has been used primarily as an educational tool in graduate and undergraduate networking and architecture courses [74, 76]. It has also been used for conducting research on a variety of networking topics including overlay multicast mechanisms [32] and peer-to-peer systems [36]. Originally, ONL was designed to give users access to four locally built extensible hardware routers [15] and a few dozen PCs for traffic generation.

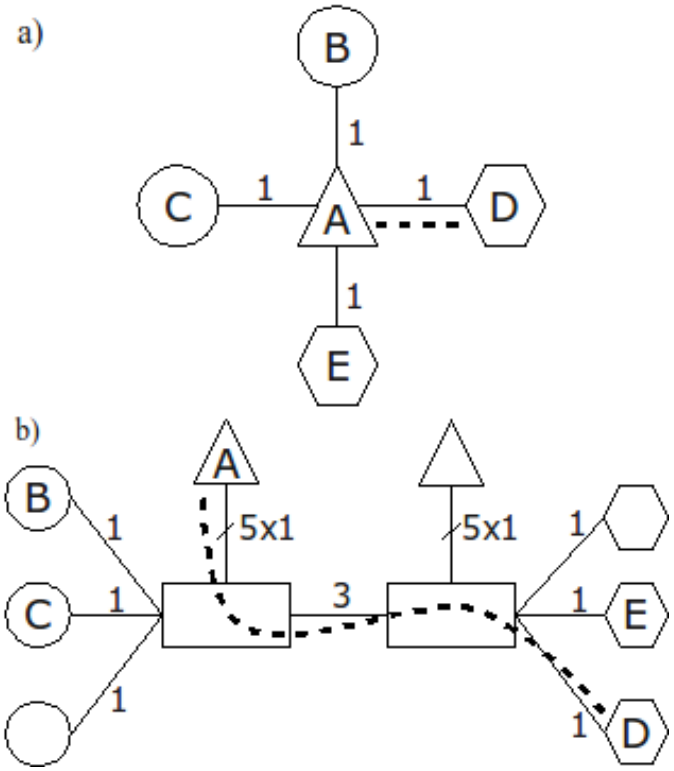

Figure 2.6: a) Example user virtual network, and b) example testbed physical network.

The basic communication framework used by early versions of ONL was similar to that shown in Figure 2.1, but none of the abstractions discussed in Section 2.2.2 existed and no thought was given to supporting heterogeneous technology or different sets of functionality on the same technology.

When it was decided to incorporate a second type of programmable router, the RLI was reworked extensively to add support for the new router (i.e., all of the configuration and monitoring menus had to be added based on the design of the router). Unfortunately, there was no clear path in the RLI or in the CRD to allow both types of routers to exist together in the same virtual topologies. So, for a time, there were actually two ONL testbeds running in parallel, one for each of the router types. This situation, along with the desire to add additional types of networking technology to the testbed, drove our initial work to extend and enhance ONL.

ONL has now been reworked from the ground up to support all of the design goals given in Section 2.1. All of the software components have been completely rewritten, and the interfaces have changed to support the framework as described in Section 2.2. The CRD uses the base type description to represent all nodes in the testbed. All reservation and configuration operations are thus type independent. The actual node states and descriptions are stored in a database, and adding new nodes or new types is a simple matter of adding new entries to the databases tables; no code has to be rewritten. The RLI now supports both the base type and specialization descriptions, building all menu and configuration options directly from the XML files for each base type and specialization. In principle, users could run any program that conforms to the CRD interface, but, practically speaking, the RLI is a very complex piece of software that would not be easy to replicate. New Base Node Daemons, and Specialization Node Daemons have been written to conform to the new interface as well. There is a template Specialization Node Daemon that can be easily modified to build new specializations. Of course, any programming language that supports socket programming can be used, but the template and associated messaging libraries are currently only available in C++.

Now that these changes are in place, the two types of routers and a small number of other resources have been combined into one testbed that forms the current ONL.

The next section provides some details about the resource types currently available in the testbed.

2.5.1 Current Resources

The Network Services Platform (NSP) [15] is the custom-built IP router from the original version of ONL. It is designed to operate in a similar fashion to larger, scalable router platforms. Each of the eight 1 Gb/s ports houses an FPGA and a general-purpose processor. The ports are connected via a 2 Gb/s cell switch. All of the standard packet processing tasks are handled by the FPGA. Users write *plugins* for the general-purpose processor to dynamically extend the router's functionality. ONL currently contains four NSPs.

The second resource type in ONL is the Intel IXP 2800 network processor [1]. IXP 2800s have seventeen cores: sixteen MicroEngine packet processing cores and one XScale management core. The particular systems used in ONL are Radisys ATCA-7010 [56] boards that have five 1 Gb/s ports for each of two IXPs. ONL currently contains seven of these boards and therefore fourteen total IXPs. The IXPs support one particular router specialization, the Network Processor-based Router, which is described fully in Chapter 3.

The NetFPGA [44] is a relatively new FPGA-based device that has already seen substantial use in both research and education. There are a number of projects readily available including IPv4 routers, Ethernet switches, OpenFlow switches [43], and packet generators [20]. Some of these projects already have specializations available in ONL, and more should be available soon. There are currently six NetFPGAs in ONL.

ONL has three different types of PCs for use as end hosts or for emulating other network technologies. The first PC type is a single-core processor with a 1 Gb/s data interface. The second PC type is a dual-core processor with a 1 Gb/s data interface. The third PC type is an eight-core processor with a 10 Gb/s data interface. Each node of these different types also has separate out-of-band control and management interface. Users are given SSH access to the control interface for every PC in their

topology for the duration of their session. A standard set of utilities and traffic generators is installed by default. Users are welcome to install other software in their user directory (which is shared across all the PCs), but users are not currently given a root shell. Any software that requires root privileges is installed by the testbed staff, and users are granted access through normal sudo mechanisms. Note that all of the PCs run kernels that allow for Click [38] to operate as a kernel module. There are currently 72 single-core PCs, 34 dual-core PCs, and 28 eight-core PCs.

All of the nodes are indirectly connected through a small number of "backbone" configuration switches. VLANs are used extensively in the switches to enforce isolation among different sessions. All of the configuration of these switches takes place automatically and invisibly from a user's perspective, via the Network Management Daemon. The reservation system is responsible for ensuring that the configuration switches have sufficient capacity to guarantee that no session could ever interfere with any other session. VLANs also provide a way for ONL to support standard Ethernet switches in experimental topologies as "virtual" resource types that do not correspond to individual pieces of hardware. This is supported only via specialized code running in the CRD.

There are currently eight backbone switches in ONL, of two different types. The first type is the Netgear GSM7352S [45], which has 48x1 Gb/s ports and 4x10 Gb/s ports for connecting to other switches. There are six of these switches in ONL. The second switch type is the Arista 7148SX [6], which has 48x10 Gb/s ports and 2x1 Gb/s ports for out-of-band management. Every user-allocatable node in the testbed is attached to one of these switches. For nodes with multiple ports, every port is connected to the same switch. One of the 1 Gb/s ports on each Netgear switch and one of the 1 Gb/s management ports on each Arista switch are connected to a physically separate management network along with the CRD. The switches are manually configured to only allow access via that separate network, thereby ensuring that users can not send configuration changes to the switches.

One possible layout of the testbed data network is shown along the middle of Figure 2.7. Indeed, this was the actual data network used in ONL for a number of years. The figure also shows how three concurrent sessions might be scheduled and mapped

to the testbed network. The top of the figure shows RLI screenshots from each session. The node and link mappings for each session are included in the testbed network diagram below the screenshot. The colors of the different types in the RLI correspond to the colors of the blocks in the diagram. Circles in the backbone switches correspond to virtual switches in the user session. For example, the left topology has one hardware router, two network processor systems, three virtual switches, and seventeen PCs. Note that the three sessions in this example are mapped to distinct subsets of the backbone switches only for clarity. In practice, different virtual networks can and often do overlap in complicated ways when mapped onto the testbed network.

2.6 Example Sessions

ONL can be used to conduct networking experiments over a wide range of areas. Three examples follow to illustrate some of these possibilities.

2.6.1 TCP Dynamics

The first example is a simple session to study TCP dynamics over a shared bottleneck link. A screenshot is shown in Figure 2.8. The topology configuration window is on the left, and two real-time charts are on the right. The 8 port device at the bottom right of the central square is an NSP. The 4 port device at the top left of the square is a NetFPGA running as an IPv4 router. The two 5 port devices at the other corners are NPRs. The small ovals are virtual switches, and the other symbols represent PCs.

Two TCP flows are started at the same time, one from a PC in the top left of the topology to a PC in the bottom right, and the other from a PC in the bottom left to a second PC in the bottom right. Network routes are configured statically so that both flows share the bottom link in the central square as a bottleneck link. The link capacity is set to 100 Mb/s. Initially, both flows share a 100 KB queue at the bottleneck. Half way through the flows, the router is reconfigured to map one of the flows to a different queue.

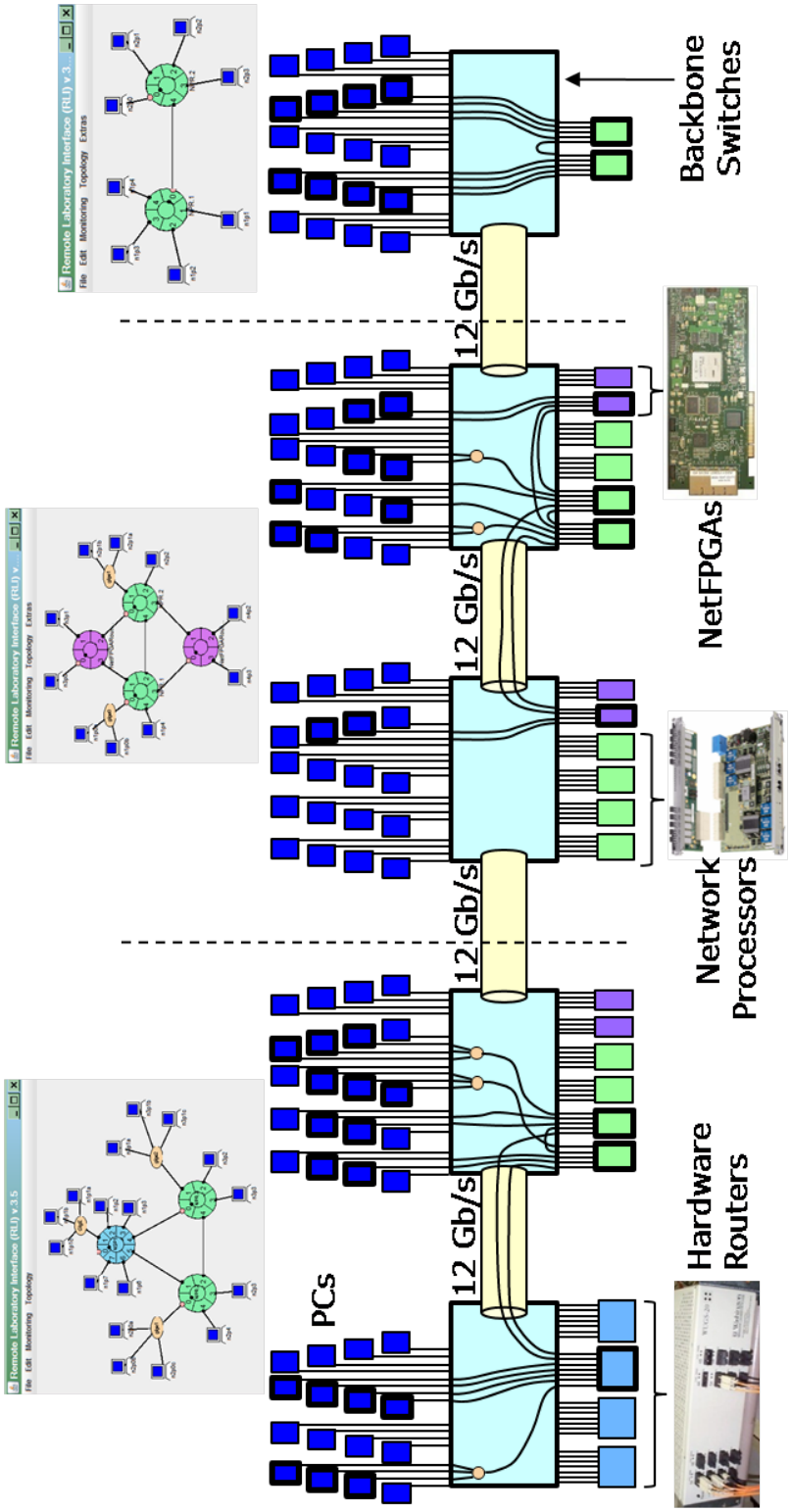

Figure 2.7: Testbed network with session mappings for three user virtual networks.

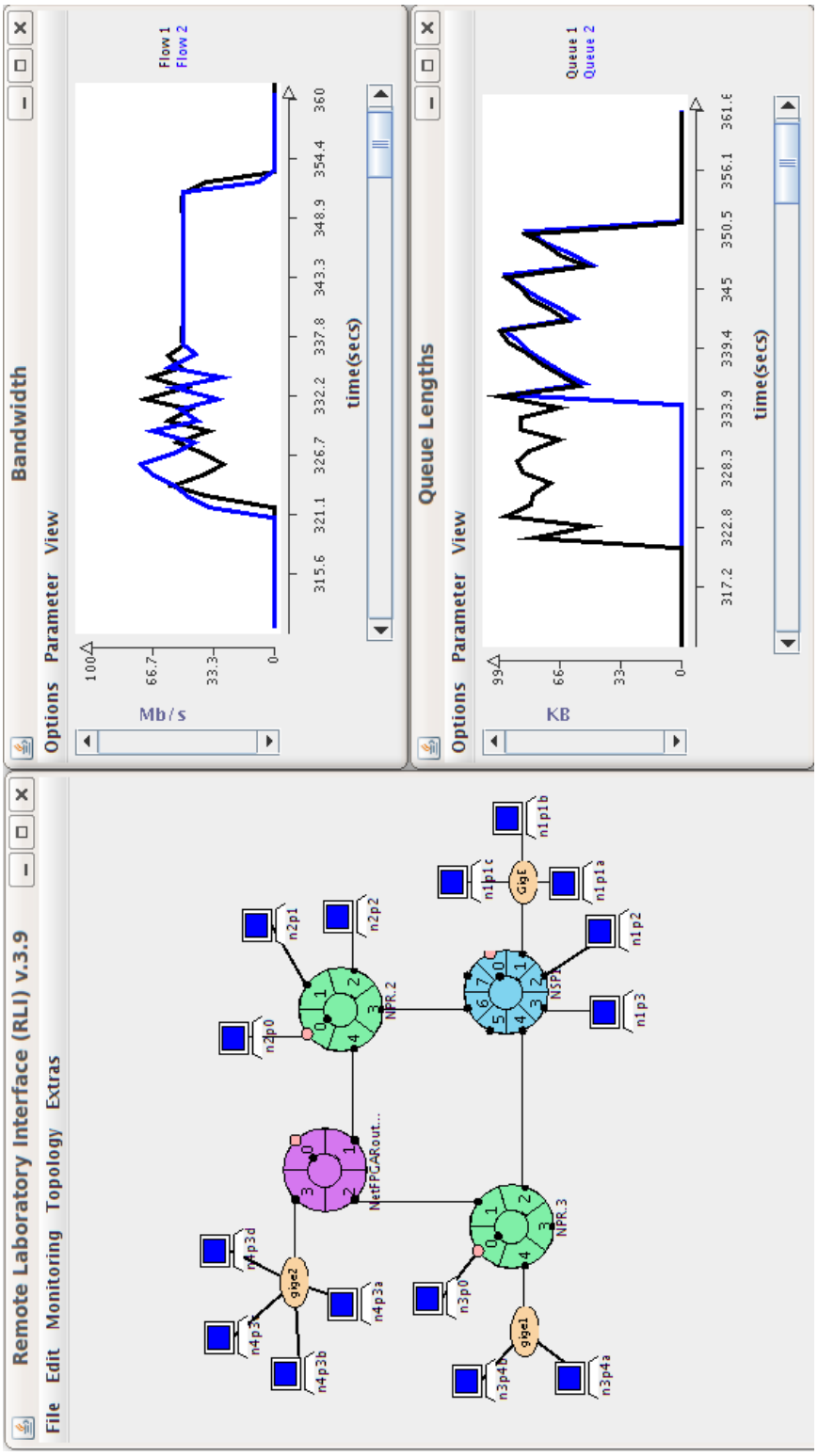

Figure 2.8: An example session in ONL studying TCP dynamics.

The results are shown on the right of Figure 2.8. The top chart shows the bandwidth of each flow, and the bottom chart shows the queue lengths of the two queues at the bottleneck. When the flows share a queue, they do not converge to their fair share of 50 Mb/s each. This follows because each flow is attempting to use packet drops as feedback to adjust its sending rate, but the shared queue leads to uneven drops for each flow. When one of the flows is remapped so that each flow is in a separate queue, the standard TCP queueing behavior is seen. Weighted Deficit Round Robin scheduling is used at the bottleneck, and each queue has the same quantum. As a result, the two flows quickly converge to their expected fair share of the link.

2.6.2 Data Center Networking

The second example uses ONL to replicate one segment of a data center network. A screenshot of the ONL topology is shown in Figure 2.9. This session uses six NetFPGAs as data center switches. In this case, they are running as OpenFlow switches, and the OpenFlow controller, NOX [30], is running on one of the PCs at the bottom of the hierarchy. This network is organized in a similar fashion to a fat tree (e.g., as in PortLand [48]), but with twice as many PCs connected to each edge switch as in a standard fat tree.

ONL allows users to build data center networks like this one quickly and easily. The result is that it is much faster to study a broad range of possibilities than it would be without a testbed infrastructure. In this example, ONL is being used to study slight modifications to fat tree topologies, but it could also be used to explore completely different data center topologies without any extra overhead.

2.6.3 Overlay Networking

The final example session is an overlay network scenario, where the overlays are built on top of a virtual network in ONL. This particular overlay network is based on the Forest [32] architecture that is designed to support highly interactive virtual worlds. Forest networks are built around provisioned tree-structured communication channels called comtrees. Comtrees support both unicast and multicast packet delivery. In a

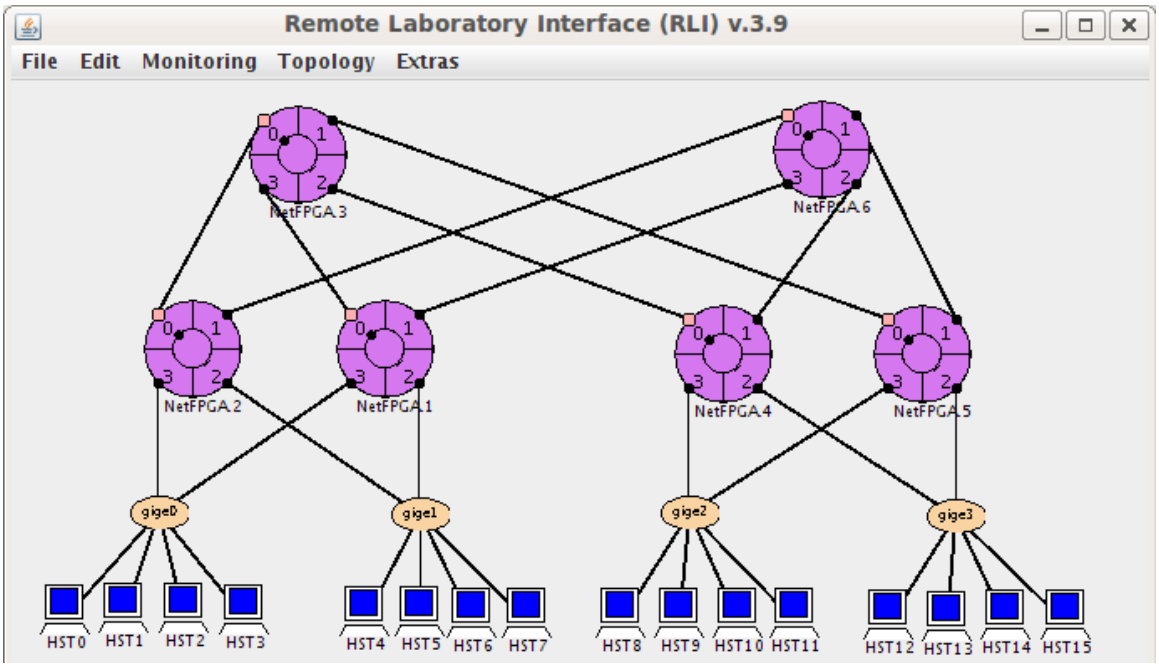

Figure 2.9: An example session in ONL studying data center networks.

typical Forest application, such as a First Person Shooter game, each comtree is associated with a separate game session or world instance. Within that comtree, then, each multicast address is associated with the state of some object in the game, such as a player's avatar. The multicast mechanism in Forest is extremely light-weight in order to support the high levels of churn in state subscription changes within such virtual worlds.

Figure 2.10 shows a screenshot of a Forest overlay operating within ONL. The four PCs in the middle of the topology implement Forest routers and are connected to one another by a full mesh of overlay links (implemented by the IP routers in the ONL context). The PCs around the periphery represent end systems in a Forest network and each of them is connected by an overlay link to one of the four Forest routers. In this sample session, the end systems are sending artificial traffic that is passing over pre-configured multicasts spanning several different comtrees. The four real-time charts in the figure show traffic rates (in packets/s) at each of the four Forest routers. The staircase packet rates are due to end systems entering and leaving comtrees at staggered times.

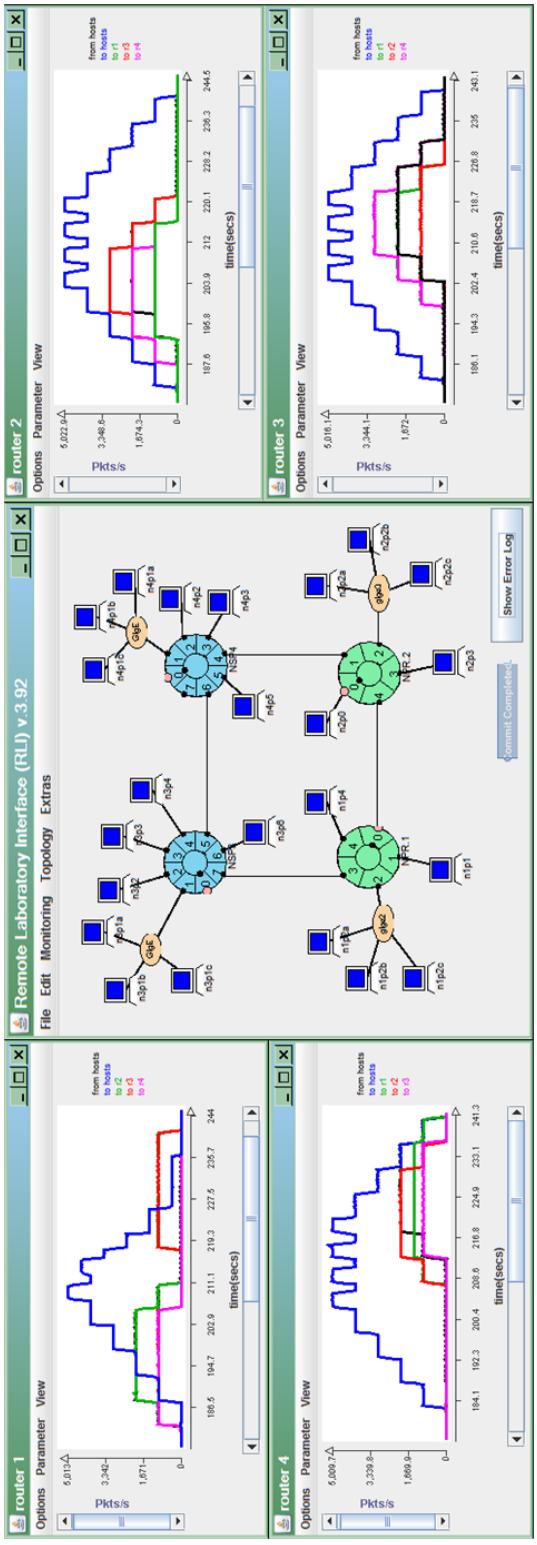

Figure 2.10: An example session in ONL for PC-based overlay networking.

2.7 Operational Details

There are a few management details that are worth noting and discussing here, as they impact the actual operation of the testbed.

Figure 2.11 is a representation of all of the software components in the testbed framework. Arrows between the various components show possible communication paths between the software components. The three distinct pieces of the CRD are shown separately, as they are three separate processes, and the Specialization Nodes Daemons are shown with dashed lines they may or may not be running on any given node. As the figure shows, the Relay routes requests from the RLIs to either the CRD or the Base Node Daemons. When the CRD contacts individual Base Node Daemons to start or stop a session, it does so directly rather than going through the Relay. This is a simplification of the original design, but a reasonable one. The CRD and Relay (and Network Management Daemon) all run on one trusted server. Therefore, routing messages from the CRD to the nodes via the Relay adds no extra security and unnecessarily overloads the Relay.

Recall from above that every node in the testbed is connected to the set of backbone switches. Recall also that most of the nodes have a separate control interface for out-of-band node access. This implies that there are two separate networks in the testbed: the "data" network for user session data connections, and the "control" network for remote access that does not interfere with the data network. In fact, there is also a third network in the testbed that is used to manage the backbone switches and other infrastructure equipment. Figure 2.12 shows how the testbed nodes and infrastructure is connected via these three networks. Each of these three networks is physically separate from the others to keep the associated traffic cleanly separated.

The figure does not show how the testbed nodes are actually connected to the backbone switches in the data network, but the detailed layout for a previous version of ONL is given in Figure 2.7. There are a small number of servers on the control network along with the control interfaces of the testbed nodes. The testbed firewall is connected to the external Internet, and filters all incoming connections to the testbed and outgoing connections from the testbed. In general, the firewall is configured to

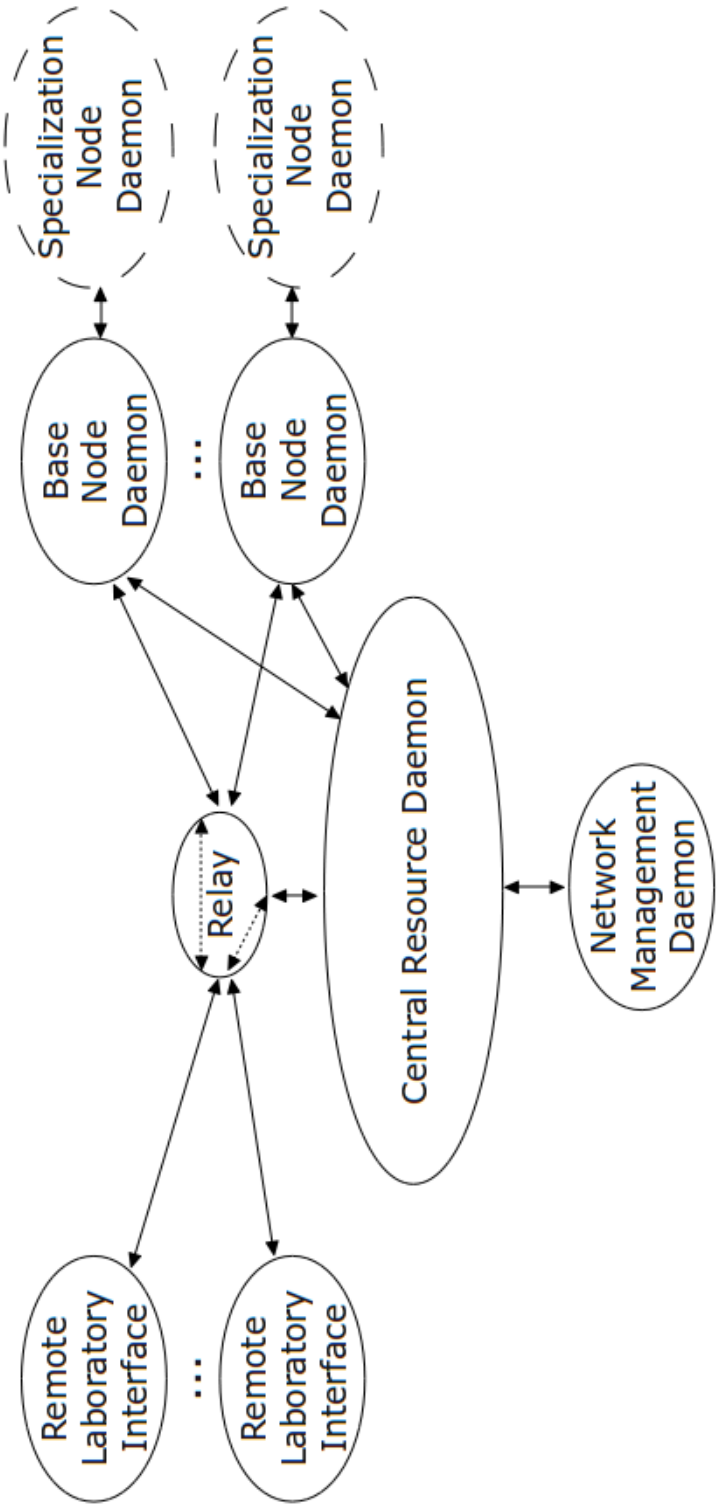

Figure 2.11: All communication paths between software components.

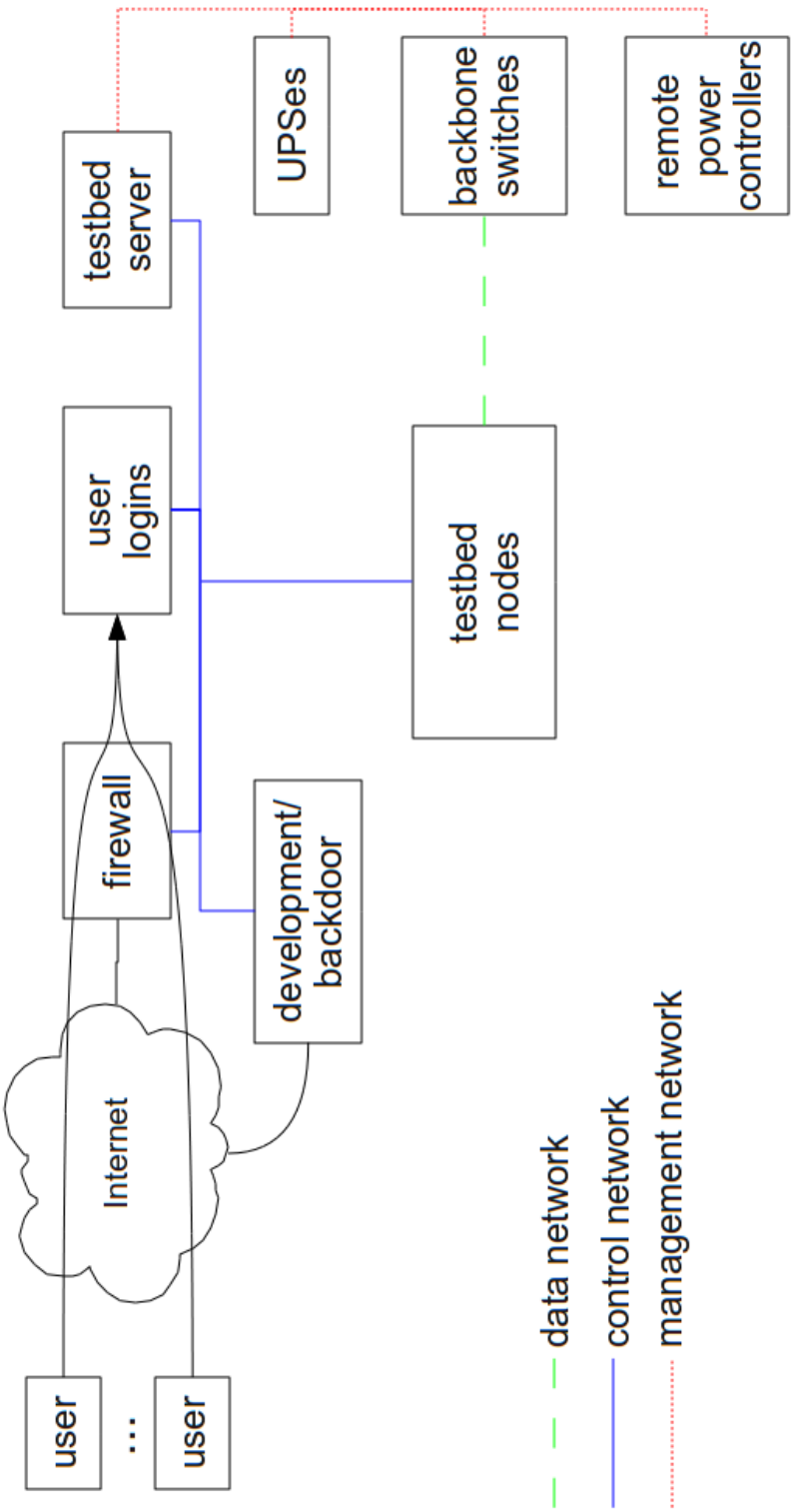

Figure 2.12: Layout of the three networks in the testbed.

be as closed as possible with particular emphasis placed on restricting outgoing traffic from the testbed. User logins over SSH are redirected by the firewall to a login server shared by all users. There is also a testbed server that runs all trusted software including the CRD. Users are not allowed access to the testbed server. There is also at least one separate development computer for building and testing testbed software. This computer is also restricted to allow only testbed staff members, and additionally serves as an extra path into the testbed control network in the case of firewall or login server failures.

The testbed server also connects to the management network. This network consists of all of the testbed infrastructure equipment that is hidden from the users. Each of the backbone switches has at least one management interface that connects to this network. All of the power-related infrastructure is attached to the management network as well. Uninterruptable Power Supplies (UPSes) are used for all power in the testbed to cover power outages. Remote Power Controllers such as the Synaccess NP-08G2 [64] are used for per-node power control. This allows nodes to be automatically power cycled if they become unreachable and provides a means to completely shutdown the testbed regardless of the current state of the nodes under extended power or air conditioning outages.

Another important operation detail is that the RLI does not contact the Relay directly. Instead, each user configures an SSH tunnel [53, 65] from their computer to the testbed. Specifically, the tunnel is terminated on the login server (where the SSH connection terminates) and then packets received from the tunnel are forward to the testbed server and the Relay particularly. All communication from the RLI and all responses to the RLI are sent over the tunnel. The two major reasons to do this are to simplify testbed and user firewall configurations (exactly one port has to be open on each side), and to encrypt all data between the RLI and the testbed. The latter is important as it allows authentication from the RLI to happen securely.

Within the testbed, each user has his or her own home area. All user home areas actually reside on the testbed server and are NFS [47] mounted over the control network on the login server. Similarly, user accounts and groups reside ultimately on the testbed server but are shared to the other servers and testbed nodes via NIS [49]. A user's home area is also exported to all nodes in their virtual network at the

beginning of each session and removed at the end of the session. In other words, only the home area for the user assigned to a node is available on that node, and only for the duration of their session. This gives users full access to all of their files on each node while minimizing the damage a user could do if they were to compromise a node. SSH access to each testbed node is also restricted to only allow the current user assigned to that node to log in. This is accomplished by dynamically adding/removing users to/from per-node access control groups at session begin and end time.

Finally, all of the PCs in ONL use a two-stage booting process. The first stage has each PC load a diskless kernel and file system over the network via PXE. The file system only needs to supply the most basic Linux environment. The last step of this first stage boot is to start a special ONL daemon that contacts the CRD to announce that this node is ready to start the second stage. The CRD responds with the location of a file system to load onto the disk of the node. There is a special directory on the testbed server that contains all the images for each PC type, and this directory is made available read-only over NFS to every node. The node then reformats the disk and writes the file system image to the disk. Many tools could be used to do this. ONL currently uses FSArchiver [25]. Once finished, a kexec is issued for the default kernel now loaded on the disk. This standard kexec system call overwrites the running kernel with the new kernel and executes it.

There are a number of benefits to using this two-stage configuration. First, every time a node reboots it is guaranteed to come back up in a clean and well-known state. Second, updating node software or configuration for a particular PC type only requires modifying and re-saving the default image. Then, as each node is rebooted at the end of every user session it will pick up the modification automatically when it reloads the default image. Third, it allows the testbed staff to build and supply multiple possible kernel or file system choices for each node. A simple dialog box on the RLI is used to choose between the available images and then that choice is passed on to node when it assigned to the user. Although ONL does not currently support it, this also provides an avenue to allow users to build and supply their own kernel images for conducting research that requires kernel modifications.

2.8 Educational Benefits

There are clear benefits to using testbeds like the Open Network Laboratory in networking and systems education. First and foremost, they help students to gain practical experience with otherwise abstract networking concepts. This is certainly true in any field, but networking technology in particular is driven by practical real world needs rather than theoretical constructs. Testbeds provide a place where students are able to start building their own insights and instincts about running systems that are very difficult to be taught directly by educators. This includes giving students a place where they can make mistakes configuring complex systems without any concern that they could hamper operations of a production network. To date, courses have been taught using ONL at the following universities: Washington University in St. Louis, University of Massachusetts Amherst, University of Kansas, DePauw University, and the Jordan University of Science and Technology.

Some efforts have been made to characterize student learning in ONL specifically [75, 76]. A few undergraduate and graduate networking courses at a small number of universities were modified to use ONL for laboratory assignments instead of the more typical simulation environment. Those courses were analyzed and compared to their predecessors in order to understand how using ONL affected the students' ability to learn fundamental networking concepts. That analysis suggested that using ONL provided better pacing to the course, which allowed the students more time to focus on the most important concepts rather than focusing on less important details. Anecdotal evidence was also collected via informal student surveys. Overall the students were much more enthusiastic about using ONL instead of a simulation program to bridge the gap between lectures and networking practice. They also generally found the testbed software easy to use and understand. Some more formal learning assessments have also been conducted, but with such small sampling sizes that it is difficult to say anything quantitatively.

The other concern when using a testbed environment is the burden on the instructor and teaching assistants. If the amount of work required from an instructor is substantially higher when using a testbed, then it is far less likely that testbeds will become more widely adopted in networking courses. The rest of this section details

an assessment done to answer this very question for the Open Network Laboratory. Note that this study was done using a previous version of the testbed.

2.8.1 Instructor Support Structure

The normal interaction between instructor (and any teaching assistants) and students is augmented by the ONL support staff who backstop the instructor. The ONL staff help the instructor to answer questions concerning seemingly strange system behavior. This approach relieves the instructors of many of the anxieties that might arise from using experimental equipment and has worked surprisingly well with the majority of questions and issues being handled by the instructor. Section 2.8.4 quantifies these interactions in detail, but a summary is shown in Figure 2.13 where the labeled arrows indicate how many emails were sent between students, instructors, and the ONL staff over one full semester. Details about courses A and B are given in Section 2.8.3. All interaction is initially handled by email and goes through the instructor who is expected to answer most of the questions. The instructor should only need occasional help from the ONL staff. Students are only allowed to contact the ONL staff directly when there is evidence of system failures.

2.8.2 Instructor Considerations

From an instructor's perspective, some aspects of using ONL in a course are no different than using any other computer system, but there are also operational issues that are somewhat unique to ONL.

First, instructors need to work out solutions to their laboratory assignments before distributing them to students. Although this is a good idea for assignments in any course, it is a particularly important precept if the instructor is also new to the virtual laboratory. The typical problem is that the instructor's mental model disagrees with ONL's conceptual model in some subtle way. One example is the link rate parameter mentioned earlier. Although common misconceptions are documented in the ONL tutorial pages and summarized in FAQs, it is easy to overlook the distinctions.

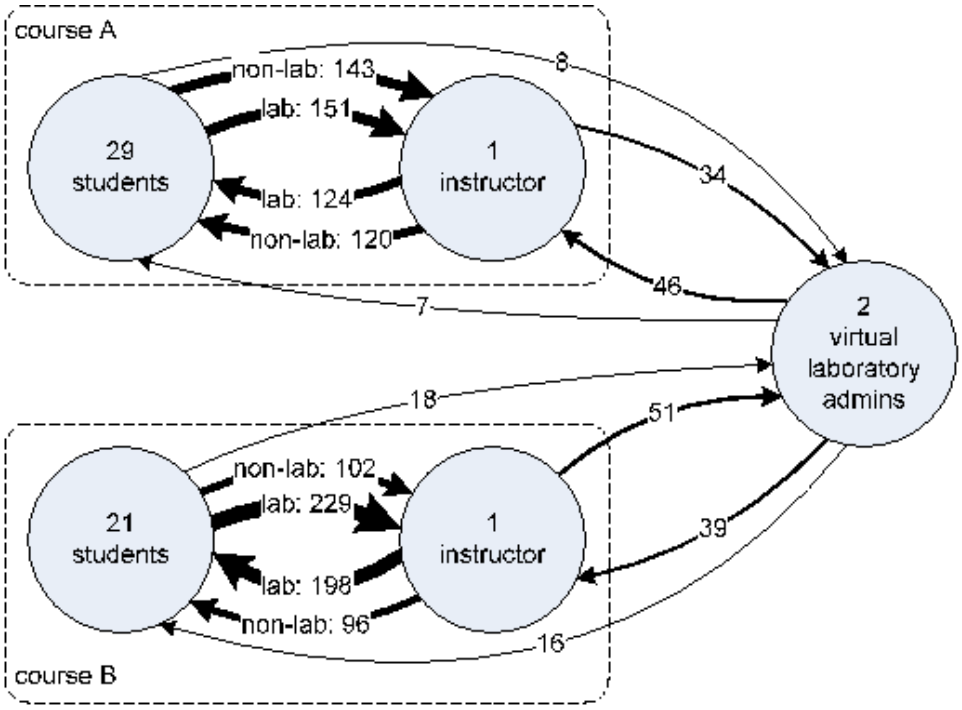

Figure 2.13: Email Interactions Over the Course of One Semester.

Second, each laboratory assignment should be preceded by paper-and-pencil exercises that emphasize the main concepts addressed by the assignment and develop approximation skills that can be used for sanity-checking experimental results. Troubleshooting exercises in which students are asked to postulate the source of unexpected measurement results and propose mini-experiments for verifying these postulates provide useful mental preparation.

Third, students should be counseled on efficient ways to conduct their experiments. For example, many of the initial RLI features can be explored without actually reserving hardware resources. A common mistake made by students in the first assignment is to reserve hardware and then spend the entire time going through the various RLI menus instead of actually using the hardware.

Fourth, there is a high level of paranoia that develops when more than one course is using the facility during the same period. "Hey, University XXX students are evil because they are using our resources" is a common complaint from students. The ONL staff has done a good job in coordinating assignment due dates to keep the

Table 2.1: Comparison of Courses That Use ONL.

	Course A	Course B
Department	ECE	CS
Only graduate students?	yes	no
Non-networking students?	yes	no
Number students	29	21
Number of labs	4	4
Labs requiring programming	0	2
Programming difficulty	n/a	basic; any language

actual contention level low, but instructors still need to pay attention and foster a cooperative attitude.

2.8.3 Courses Studied

Our observations in this section are based on two courses taught in 2007, Course A and Course B, conducted at different universities as shown in Table 2.1. The main differences between these courses are:

- Course A is aimed specifically at giving breadth to graduate students that may not be majoring in networking.
- Course B requires students to write basic socket programs to transmit UDP packets. The course survey indicated that the programming was not a significant hurdle because they were allowed to use any language.

2.8.4 Interaction Data

Data was collected on the amount of email interaction between students, instructors, and ONL staff as indicators of the amount of work accompanying laboratory assignments for instructors and support staff. The data is normalized to a class of 29 students, the size of Course A.

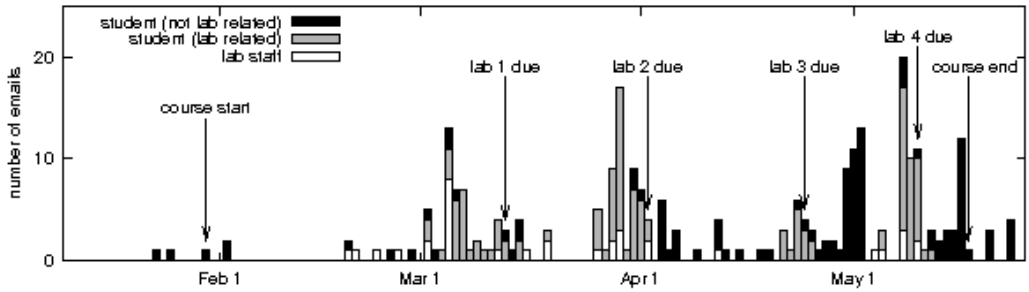

Figure 2.14: Email Counts in Course A (With ONL).

Figure 2.14 shows some of the data for Course A during the semester. It shows the number of daily emails from the instructor to students and ONL staff (lab staff) with a distinction between emails that were and were not related to a lab assignment. One observation is that there was very little interaction with ONL staff except near the beginning of the first assignment, i.e., the instructor handled most of the questions. The increased interaction with ONL staff during the week of March 1 was due to an overly aggressive security policy that resulted in the blocking of connections from student hosts that fit the profile of attack activity.

It is interesting to note that there was little interaction in the third assignment. The assignment involved exploring network behavior due to the competition between TCP and UDP flows and designing/performing an experiment to prove or disprove a hypothesis (e.g., UDP achieves a higher utilization on the bottleneck link than TCP). In fact, this assignment generated the most student interest: they were excited and pursued the problem on their own (without checking with the instructor).

To give some perspective to Figure 2.14, Figure 2.15 shows the same type of email interaction data for the same course taught previously when assignments involved socket programming and ns-2 [50] simulations instead of ONL. In this figure, *lab staff* refers to the local computer support staff, not the ONL staff. At first glance, the data indicates that instructors can expect to be interrupted by students less than if they were using ONL. Actually, this observation seems reasonable for two reasons:

- There were fewer operational issues. Students asked fewer questions on how to write socket programs. With ONL, there were operational issues; e.g., using `SSH` to access ONL resources.

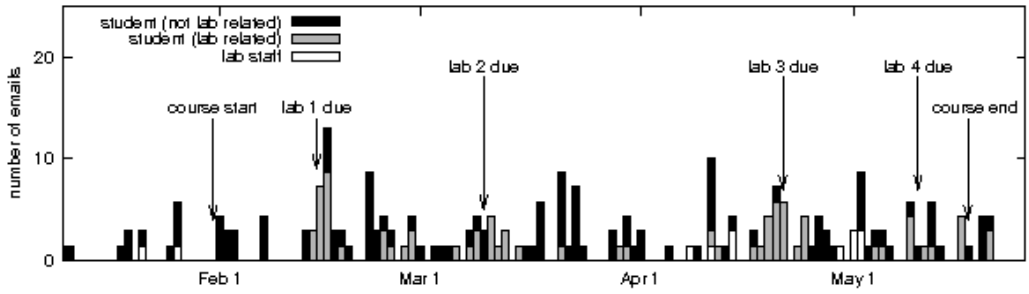

Figure 2.15: Email Counts in Course A (Without ONL).

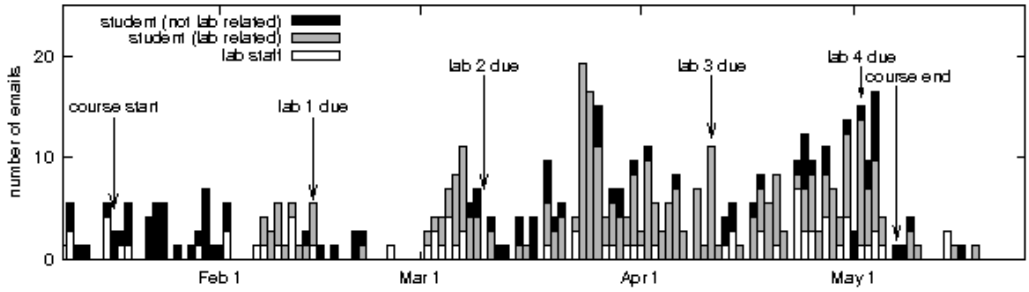

Figure 2.16: Email Counts in Course B (With ONL).

- There were fewer questions regarding what to do in socket programming assignments. With ONL, there were more questions regarding the precise meaning of an assignment's requirements and how to meet those requirements.

The second point is worth elaborating. In the non-ONL course, there may have been more improper collaboration between students. Since students did not have to log into the system and set up their own experiment, they may have solved projects jointly and thus had peers to help and answer questions. In the ONL version of the course, peers could not really provide much help when something did not seem to behave as expected and had to work through a problem until they required expert advice.

Informal discussions with the teaching assistant in Course B corroborated this observation. Furthermore, there was a clear evolution in the type of student-instructor interactions as the semester progressed. Interactions at the beginning of the course

concerned more operational ONL/RLI issues as the students were familiarizing themselves with a foreign environment. Towards the end of the course, the student questions evolved into discussions focused more on fundamental networking concepts than on operational issues.

Another important statistic is that the volume of email shown in Figure 2.14 (the ONL case) that is above that shown in Figure 2.15 (the non-ONL case) is not particularly significant. The benefit of increased student engagement and thinking precision far outweighs the small increase in student-instructor interaction time.

Figure 2.16 shows the interaction data for Course B. The most significant difference with Course A is that there is interaction on almost every day during the last two assignments. Furthermore, there is more interaction with ONL staff than in Course A. This increase in interaction can be explained by noting that these two assignments involved developing router plugins.

Developing a router plugin is almost equivalent to writing a module in C for an embedded processor where the normal debugging facilities are absent. The plugin environment delivers the packets to the user plugin in network byte order with attached packet headers. Errant code can hang the plugin processor requiring the student to abandon an experiment and restart – a time consuming process. As such, writing a plugin is initially daunting to students who are used to interactive debugging tools, simple data formats, and rapid retries.

Note that a shortcoming of the email counting approach is that it does not distinguish between those emails that were caused by ONL problems and those that were just about the assignment.

2.8.5 Observations

In closing, some observations from discussions with the course instructors, teaching assistants, and the ONL staff are given.

It would be interesting to collect data in classes that are dominated by undergraduate students. Graduate students are generally more knowledgeable and more capable of

coping with non-ideal situations. On the other hand, undergraduate students are less likely to struggle through such situations and ultimately complete assignments.

It is clear that good ONL assignments focus graduate students on fundamental networking principles. In fact, laboratory assignments give all students an opportunity to reflect on fundamental concepts and to see them in practical terms. Bringing together theory and practice in this way is certainly one of the main strengths of using a virtual laboratory for assignments.

It appears that with graduate students, the ONL assignments have not promoted peer-to-peer interaction. One teaching assistant said: "Many graduate students don't do peer-to-peer instruction. They want to figure it out themselves. And if they can't, they're going straight to an expert." Peer-to-peer interaction is difficult to quantify although cooperative approaches to learning appear to correlate with improved student satisfaction and superior outcomes. However, there is nothing preventing an instructor from using an ONL experiment running in real time to initiate a discussion about a fundamental concept.

Chapter 3

Specialized Packet Processing Platforms

The previous chapter described a new testbed framework that supports node heterogeneity and extensibility at a fundamental level. The primary driver for this is to support research and educational objectives that can take advantage of complex networking technology. Such technology can often be quickly reconfigured to fill many different roles in the network. Configuring these types of complex devices can be daunting to users, particularly when multiple different types of technology are being used in conjunction. To that end, the testbed framework also includes an extensible user interface that provides users with a consistent and intuitive view of their virtual networks.

This chapter presents a series of examples that illustrate the extensibility of the testbed framework. Specifically, the examples come from testbed nodes currently in the Open Network Laboratory, which is a prototype implementation of the general testbed framework. Each example includes a description of the networking device in question along with one or more of the specializations that are available to users in ONL.

Three types of nodes will be discussed. The first example is a standard PC type with a single simple specialization for normal end host behavior. The second example uses the NetFPGA platform developed at Stanford University [27, 44]. Three specializations are described that reflect three different existing NetFPGA projects that have been ported to the ONL environment. Finally, the third example is a network processor system based on Intel IXP 2800s [1]. This last example contains a

detailed description of one particularly complex IPv4 router specialization, the Network Processor-based Router [71], which was developed as part of this dissertation research.

3.1 PC

The first example platform is one of the PC types available in ONL. Namely, this PC type has two quad-core Intel Nehalem [35] processors, 12 GB of memory, one 10 Gb/s data interface, and one 1 Gb/s control interface. The base description for this type is given first, followed by one simple specialization.

3.1.1 PC Base Type

The ONL base description for this PC type is shown in Listing 3.1.

Note that this description is substantially different from the example base description shown in Figure 2.2. That example description illustrated the bare minimum needed for the generic testbed framework. The base description shown here in Listing 3.1 is the actual one used by the RLI in the current version of the Open Network Laboratory. As such, there are additional fields that are used by the RLI for various purposes. Each of the fields in the listing will now be discussed.

Line 2 of the listing declares that this is base type, or "hardware," description named "PC8core." There is also a version attribute for supporting different versions of the same type. Line 3 is an additional field necessary in the ONL environment that denotes whether or not this type is part of a "cluster." In this context, clusters are types that have some physical inter-dependencies and therefore must be allocated together. The PC8core type has no such dependencies, and so the field is set to "false." The "daemon" field gives the default TCP port number for the Base Node Daemon associated with this type. The port number for this type is 3551, while a value of "-1" would indicate that there is no Base Node Daemon for this type.

```
<?xml version="1.0"?>
<hardware typeName="PC8core" version="1">
  <clusterOnly>false</clusterOnly>
  <daemon>3551</daemon>
  <componentType>host</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
    </param>
  </init>
  <commands/>
  <monitoring>
    <command opcode="62" numParams="2">
      <displayLabel>UserData</displayLabel>
      <description>user data read from file</description>
      <units>unknown</units>
      <param editable="true">
        <plabel>file</plabel>
        <ptype>string</ptype>
      </param>
      <param editable="true">
        <plabel>field</plabel>
        <ptype>int</ptype>
      </param>
    </command>
    <command opcode="63" numParams="2">
      <displayLabel>UserDataTS</displayLabel>
      <description>user data read from file with time stamp</description>
      <units>unknown</units>
      <param editable="true">
        <plabel>file</plabel>
        <ptype>string</ptype>
      </param>
      <param editable="true">
        <plabel>field</plabel>
        <ptype>int</ptype>
      </param>
```

Listing 3.1: (a) PC8core.hw; the base description for the 8-core PCs currently available in ONL.

```
42      </command>
43    </monitoring>
44    <ports numPorts="1" interfaceType="10G">
45      <fields/>
46      <commands/>
47      <monitoring/>
48    </ports>
49  </hardware>
```

Listing 3.1: (b) PC8core.hw; the base description for the 8-core PCs currently available in ONL.

The "componentType" field is used by the RLI for two reasons: graphical node representation and automatic IP address assignment. The valid values are "host," "switch," "router," and the special value "any." Each of the three non-special values has a different graphical icon in the RLI to facilitate easy recognition of different types in a large network. The RLI also uses these different values in its automatic IP address and subnet assignments. As the user builds their topology in the RLI, IP subnets and addresses are calculated and assigned automatically to every node. Of course, the algorithm that handles these assignments needs to know if each node is something that extends, participates in, or terminates a subnet. The special "any" value dictates that this type of node could be used in any of the three roles and that there isn't a clear reason to default to one role over the others. In such a case, the RLI automatically generates an additional menu that forces the user to choose which role each node of this type will fill when the node is initially added to their topology. The PC8core type is set by default to act as a "host."

Lines 6 through 43 are actually sections that are used primarily in specialization descriptions. That is, they contain information related to configuration and monitoring. In base descriptions such as this one, this means configuration and monitoring commands that are independent of any specializations and can be used whether or not there is an active Specialization Node Daemon.

Line 6 shows an empty "fields" section. Although unused here, this section can be used to declare a set of parameters that are shared among multiple tables or commands in the rest of the description. Similarly, line 15 shows an empty "commands" section, which means that there are no configuration commands for the PC8core base type.

Lines 7 through 14 define the "init" section. This section is used to list any initialization parameters that could be set and passed to the Base Node Daemon during node set up. The PC8core type has only one parameter. The "UserDaemonPath" parameter is used to pass the file system location of the Specialization Node Daemon, if any. It is declared in the base type description and defined specifically in the specialization description. Lines 10 through 13 describe the actual parameter that is sent to the Base Node Daemon. Each parameter has an "editable" attribute that specifies whether or not the user is allowed to modify the parameter. The "plabel" field is just a label string that is displayed to the user when they are editing or viewing this parameter, and the "ptype" field indicates what type of parameter this is (string, integer, boolean, etc). If there were a default value for this parameter, it would be given in a "default" field after the "ptype" field. Again, in the PC8core base type and other base types there is no default value, but the value must be given in each specialization description.

Lines 16 through 43 define two monitoring commands that are supported by the Base Node Daemon. The two commands are nearly identical, so only the first will be discussed in detail. Line 17 declares a monitoring command that requires two parameters. There is also an "opCode" attribute that is a unique identifier used by the Base Node Daemon to identify this command. The "displayLabel" field contains the string label that the user will see in the monitoring menu for this command, in this case "UserData." This "UserData" command is used to monitor data that the user writes to a file. The Base Node Daemon periodically reads the file and returns the values to the RLI to be displayed on real-time charts. The two parameters needed to handle this command are given on lines 21 through 28. The first parameter is a string parameter for the name of the file with the user's data, and the second parameter is the field or column number within the file to read. That is, the user's data file may have multiple columns of data, and each monitoring command sent could read a different column. In this way, users are able to easily monitor data produced by a node without the need to supply a Specialization Node Daemon.

Finally, lines 44 through 48 describe the network interfaces for the PC8core base type. There is only one interface, as indicated by the "numPorts" attribute. The "interfaceType" attribute is used to specify the type of the interface. ONL currently has two types of interfaces: "1G" and "10G," corresponding to 1 Gb/s Ethernet and

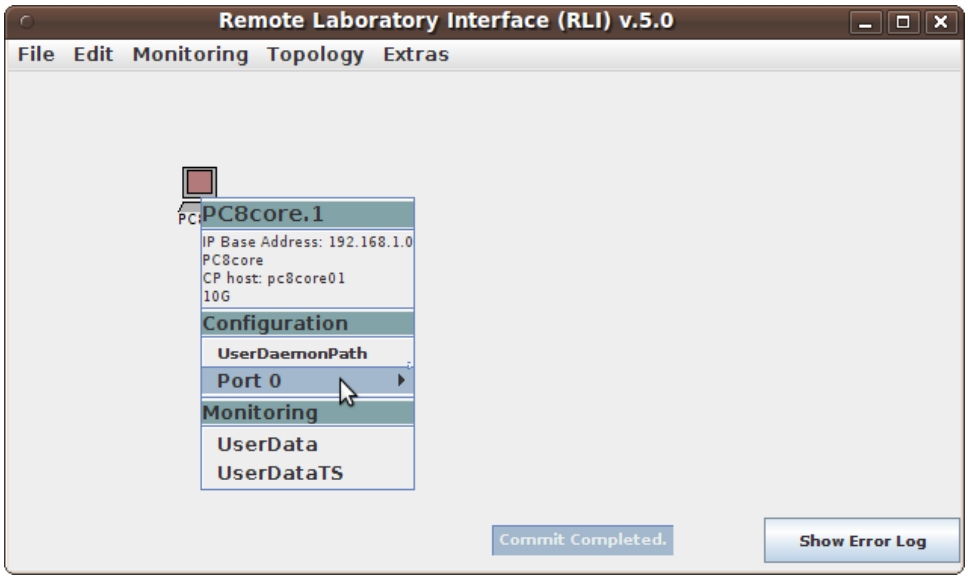

Figure 3.1: RLI screenshot of the configuration menu for the PC8core type.

10 Gb/s Ethernet, respectively. The single interface on the PC8core type is a 10 Gb/s interface. Note that the ONL infrastructure currently assumes that all network interfaces on one node are of the same type. Lines 45 through 47 define the per-port fields, configuration commands, and monitoring commands, respectively. In this case, there are no per-port commands of any kind.

Figure 3.1 is an RLI screenshot of a topology with a single PC8core node. In the figure, the user has clicked on the PC icon to open the configuration and monitoring menu. The top of the menu lists some useful information for the node, including the node's logical name, IP addresses, interface type, etc. The next menu section lists all of the configuration commands, including the initialization command. In this screenshot, the user has highlighted the "Port 0" menu option, which would normally expand into a second menu with all of the per-port configuration options but in this case shows nothing as there are no per-port commands. Finally, the third menu section contains both monitoring commands that were defined in Listing 3.1.

3.1.2 PC Host Specialization

Although PCs are certainly capable of emulating most networking devices, they are most commonly called on to act as standard end hosts in the Open Network Laboratory. As such, all the PC types available in ONL have end host specializations available. The end host specialization for the PC8core base type described above is shown in Listing 3.2.

Line 2 of the listing declares that this is a specialization, or subtype, description with the name "Host." Line 3 uses the "hwType" section to denote that this is a specialization of the PC8core base type, and the "resource" field on line 4 lists specifically the file name of the XML description for the base type. All specializations must have these declared and have the base type file available.

The rest of the specialization description file uses the same syntax and structure as the base description file. The specialization inherits everything from the base type, much as happens with class-based inheritance in object oriented programming languages. All fields and commands in the base type are also available in the specialization. Any fields or commands can be replaced or overridden with new values in the specialization description.

For example, lines 8 through 16 of Listing 3.2 override the "UserDaemonPath" initialization parameter that was originally defined in the base description file. In this case, everything is left unchanged with the exception that the specialization provides a default value (line 14) for the parameter. This path, "/users/onl/pc8core/host" is the full path on the ONL file system to the Specialization Node Daemon for this specialization.

The other new section to note in this specialization is the route table, which is described on lines 24 through 108. All tables have two sets of fields, configuration commands, and monitoring commands: one set that is table-wide and one that is per-entry in the table. The route table for this Host specialization has no table-wide commands of any kind. Line 28 starts the "entry" section, which describes everything about one table entry. Every table entry has two representations: data and display. The data representation is the true form of the table, where every table field is delineated and treated separately. The RLI also supports a display representation that

```
<?xml version="1.0"?>
<subtype typeName="Host" version="1">
  <hwType typeName="PC8core">
    <resource>PC8core.hw</resource>
  </hwType>
  <componentType>host</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
      <default>/users/onl/pc8core/host</default>
    </param>
  </init>
  <commands/>
  <monitoring/>
  <ports>
    <fields/>
    <commands/>
    <monitoring/>
    <tables>
      <routeTable title="RouteTable">
        <fields/>
        <commands/>
        <monitoring/>
        <entry name="route" numFields="3" numColumns="2" enable="false">
          <fields>
            <field editable="true">
              <plabel>prefix</plabel>
              <ptype>ipaddress</ptype>
              <default>0.0.0.0</default>
            </field>
            <field editable="true">
              <plabel>mask</plabel>
              <ptype>int</ptype>
              <default>32</default>
            </field>
            <field editable="true">
              <plabel>nexthop</plabel>
              <ptype>nexthop</ptype>
              <default>0</default>
            </field>
```

Listing 3.2: (a) PC8core-Host.shw; the end host specialization description for an 8-core PC.

```
          </fields>
          <commands>
            <addCommand opcode="73" numParams="3">
              <displayLabel>AddRoute</displayLabel>
              <description>add_route</description>
              <param editable="true">
                <plabel>prefix</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="true">
                <plabel>mask</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="true">
                <plabel>nexthop</plabel>
                <ptype>field</ptype>
              </param>
            </addCommand>
            <deleteCommand opcode="75" numParams="3">
              <displayLabel>DeleteRoute</displayLabel>
              <description>delete route</description>
              <param editable="false">
                <plabel>prefix</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>mask</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>nexthop</plabel>
                <ptype>field</ptype>
              </param>
            </deleteCommand>
            <command opcode="74" numParams="3">
              <displayLabel>Update Nexthop</displayLabel>
              <description>update route's next hop</description>
              <param editable="false">
                <plabel>prefix</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>mask</plabel>
                <ptype>field</ptype>
              </param>
```

Listing 3.2: (b) PC8core-Host.shw; the end host specialization description for an 8-core PC.

```
              <param editable="true">
                <plabel>nexthop</plabel>
                <ptype>field</ptype>
              </param>
            </command>
          </commands>
          <monitoring/>
          <display>
            <column title="prefix/mask" width="120">
              <fieldName>prefix</fieldName>
              <symbol>/</symbol>
              <fieldName>mask</fieldName>
            </column>
            <column title="nexthop" width="50">
              <fieldName>nexthop</fieldName>
            </column>
          </display>
        </entry>
      </routeTable>
    </tables>
  </ports>
</subtype>
```

Listing 3.2: (c) PC8core-Host.shw; the end host specialization description for an 8-core PC.

is used purely to customize how the tables are actually displayed to the user. The "numFields" entry attribute declares how many actual fields there are in the table, and the "numColumns" declares how many columns will be displayed to the user.

The Host specialization route table has three data fields: route prefix, route mask, and next hop. Fields are described in exactly the same way as parameters, i.e., they have a "plabel" display name, a "ptype" variable type, and a "default" value. In this case, the "prefix" field uses the special type "ipaddress," which is a string representation of an IP address in standard dotted decimal notation. The "mask" is a normal integer, and the "nexthop" is another special type for passing next hop information that often includes both an output port number and a next hop IP address. These three fields are referred to explicitly in the commands that follow their declaration.

Route table entries for this specialization also have three per-entry commands. In fact, all tables must define at least two per-entry commands: the add command and the delete command. The RLI looks for these commands specifically within the entry section for each table. Zero or more other commands can be added that use or modify an existing table entry. As is often the case, the add and delete commands for the route table in the Host specialization involve every field in the table entry. For example, lines 50 through 53 describe the prefix parameter to be sent to the Specialization Node Daemon with every add route command. Notice that the "ptype" is "field", meaning that the parameter refers to a previously defined field with the same "plabel" as the parameter. The third per-entry command is the "Update Nexthop" command, which is used to modify the "nexthop" value in an existing table entry. The Specialization Node Daemon requires all three fields in order to correctly execute the update command, so all three are once again included as parameters. Also note that the first two parameters, the "prefix" and the "mask" are not editable, but the "nexthop" parameter is editable by the user. The result is that when a user selects a route table entry and chooses the "Update Nexthop" command, the RLI generates a dialog box that shows the values of the "prefix" and "mask" in a non-editable box. The "nexthop" is, of course, in an editable box in the dialog. This ensures that the user only modifies the "nexthop" value as intended for the command.

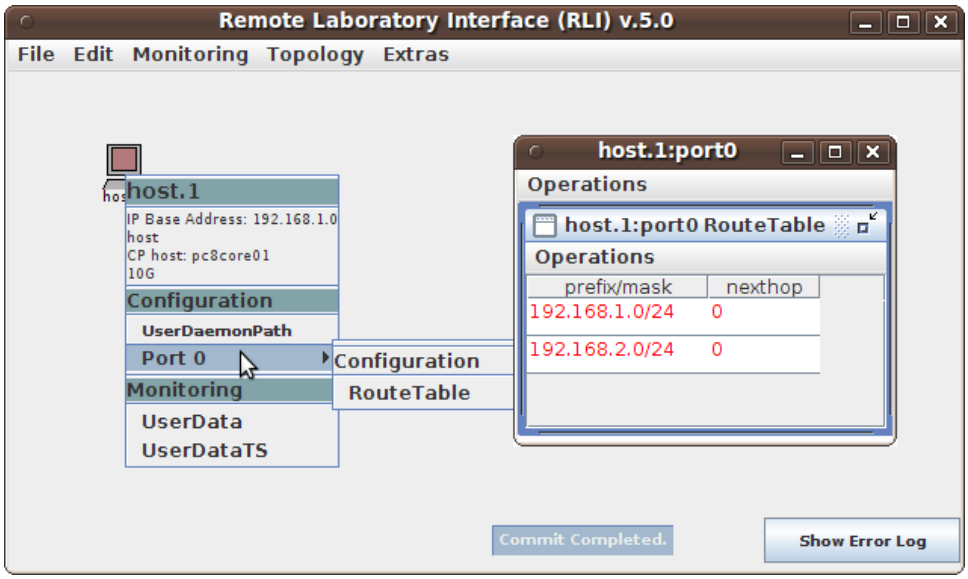

Figure 3.2: RLI screenshot of the configuration menu for the Host specialization of the PC8core type.

Finally, the "display" section for the route table is contained on lines 97 through 106. This section defines how the user will actually see each entry in the table. Recall from line 28 that there are two columns in the displayed table. Each column has a title defined in the "title" attribute for the column section, and consists of a series of one or more fields and symbols. This table combines the "prefix" and "mask" fields into a single column separated by a "/" thereby resulting in the standard CIDR notation for a route.

Figure 3.2 is an RLI screenshot of a topology with a single node of this specialization type. It is nearly identical to Figure 3.1. The only difference between the two is that now there is a route table defined with the per-port configuration options for the node, as expected from Listing 3.2. Note that the monitoring commands are available in the specialization even though they were defined only in the base description. The figure also contains a second window showing the actual route table for the node. This particular route table shows two routes each with a nexthop that indicates all matching packets should be sent out port 0.

To complete this example, all of the source code for the Specialization Node Daemon for the Host specialization is provided in Appendix A. There are 6 C++ source files that together contain 421 lines of code (including blank lines, comments, etc)

and that comprise all of the user-written code for this Specialization Node Daemon. Every Specialization Node Daemon also relies on a shared C++ library that simplifies the details of the messaging protocol and other processing tasks. The library is provided to the user, and there is no need for the user to modify or even have a deep understanding of the library source code.

3.2 NetFPGA

NetFPGAs [41] are particularly versatile networking platforms that can be used as routers, switches, traffic generators, and anything else that NetFPGA developers can build. The platform itself is a PCI card with a medium-sized FPGA (Xilinx Virtex II Pro [77]) and 4x1 Gb/s network interfaces. Developed at Stanford University, the NetFPGA has been widely adopted in both networking research and education. It has also been available in the Open Network Laboratory for some time [73].

This section will describe how the NetFPGA can be used in ONL to fill three different roles in the network. Each of these three examples relies on pre-existing NetFPGA hardware designs and software. That is, no new NetFPGA functionality has been added in order to build the specialization description and Specialization Node Daemon. Before these specializations are discussed, the base description for the NetFPGA is given.

3.2.1 NetFPGA Base Type

The base description for the NetFPGA is shown in Listing 3.3.

The NetFPGA base description is nearly identical to the one for the 8-core PC in Listing 3.1. Indeed, only three lines in the description are different. Line 2 specifies the type name to be "NetFPGA" rather than "PC8core." Line 5 sets the default "componentType" to "any." Recall that this special value for the "componentType" means that each node of this type could act as a "host," "switch," or "router." The RLI will thus ask the user to choose which of these behaviors to use as each NetFPGA node is added to their topology. Lastly, line 44 indicates that this type has four 1 Gb/s

```
<?xml version="1.0"?>
<hardware typeName="NetFPGA" version="1">
  <clusterOnly>false</clusterOnly>
  <daemon>3551</daemon>
  <componentType>any</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
    </param>
  </init>
  <commands/>
  <monitoring>
    <command opcode="62" numParams="2">
      <displayLabel>UserData</displayLabel>
      <description>user data read from file</description>
      <units>unknown</units>
      <param editable="true">
        <plabel>file</plabel>
        <ptype>string</ptype>
      </param>
      <param editable="true">
        <plabel>field</plabel>
        <ptype>int</ptype>
      </param>
    </command>
    <command opcode="63" numParams="2">
      <displayLabel>UserDataTS</displayLabel>
      <description>user data read from file with time stamp</description>
      <units>unknown</units>
      <param editable="true">
        <plabel>file</plabel>
        <ptype>string</ptype>
      </param>
      <param editable="true">
        <plabel>field</plabel>
        <ptype>int</ptype>
      </param>
```

Listing 3.3: (a) NetFPGA.hw; the base description for the NetFPGA.

```
42      </command>
43    </monitoring>
44    <ports numPorts="4" interfaceType="1G">
45      <fields/>
46      <commands/>
47      <monitoring/>
48    </ports>
49  </hardware>
```

Listing 3.3: (b) NetFPGA.hw; the base description for the NetFPGA.

ports instead of the one 10 Gb/s port available for the 8-core PC. Otherwise, all the initialization parameters and monitoring commands are the same for the NetFPGA as for the 8-core PC.

3.2.2 NetFPGA Ethernet Switch Specialization

The first example specialization for the NetFPGA is an Ethernet switch. Standard Ethernet switches require no special configuration, and the NetFPGA version is no different. As such, the specialization is relatively simple. There are no configuration commands and the only monitoring commands are per-port packet counters. Listing 3.4 is the specialization description for the NetFPGA Ethernet switch.

The associated Specialization Node Daemon simply loads the existing Ethernet switch code onto the NetFPGA at start time. It then reads counters on the NetFPGA to get the per-port packet receive and transmit counts.

The top of Figure 3.3 shows an Ethernet network topology with 5 NetFPGAs acting as Ethernet switches, 5 virtual Ethernet switches, and 25 PCs. The 5 small ovals are the virtual Ethernet switches and the 5 larger circles are the NetFPGAs. The PCs in this case are single-core PCs running an end host specialization similar to the one described above for 8-core PCs. The bottom of the figure shows a chart monitoring packet rates through the Ethernet switch in the middle of the topology when there are many flows traversing the network.

```
<?xml version="1.0"?>
<subtype typeName="Ethernet Switch" version="1">
  <hwType typeName="NetFPGA">
    <resource>NetFPGA.hw</resource>
  </hwType>
  <componentType>switch</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
      <default>/users/onl/netfpga/ethernet_switch</default>
    </param>
  </init>
  <commands/>
  <monitoring/>
  <ports>
    <fields/>
    <commands/>
    <monitoring>
      <command opcode="100" numParams="0">
        <displayLabel>RXPKT</displayLabel>
        <description>read rx packet count</description>
        <units>packets</units>
      </command>
      <command opcode="101" numParams="0">
        <displayLabel>TXPKT</displayLabel>
        <description>read tx packet count</description>
        <units>packets</units>
      </command>
    </monitoring>
  </ports>
</subtype>
```

Listing 3.4: NetFPGA-EthernetSwitch.shw; Ethernet switch specialization for the NetFPGA.

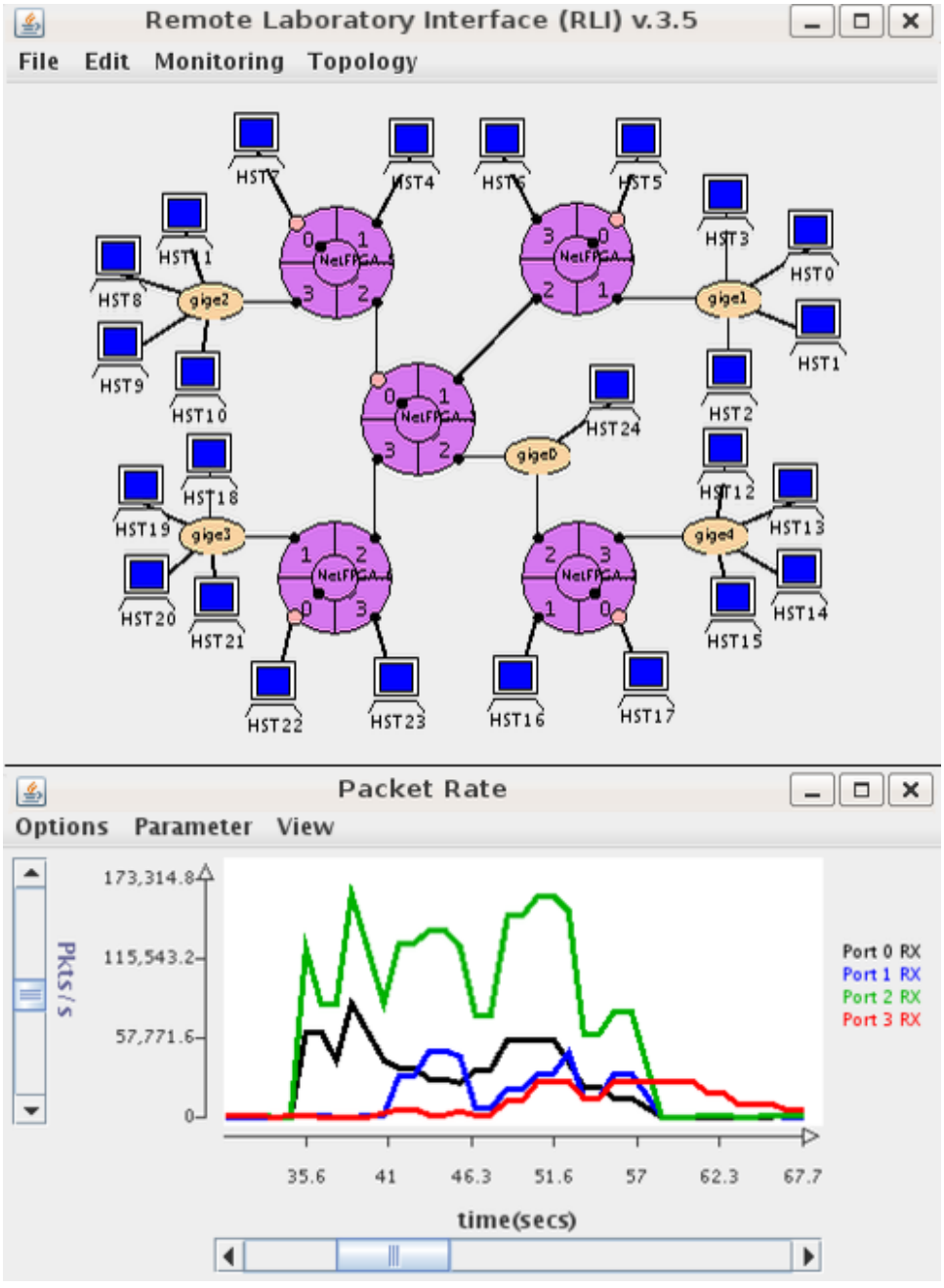

Figure 3.3: An Ethernet network with real-time packet counts from one NetFPGA Ethernet switch.

3.2.3 NetFPGA IPv4 Router Specialization

The second example NetFPGA specialization is an IPv4 router. The specialization description is given in Listing 3.5. This specialization is remarkably similar to the host specialization for the 8-core PC. The only significant difference is that the "componentType" is set to "router" here, instead of "host."

Figure 3.4 shows an example using this specialization that might be seen in an introductory networking course. The user's topology is shown on the left of the figure. The 4 port device at the bottom right of the central "square" is a NetFPGA running the IPv4 router specialization. The 8 port device above it is an NSP and the 5 port devices to the left are NPRs. The small ovals connected to each of the routers are virtual Ethernet switches. The other symbols are hosts.

In this example, the user is studying TCP dynamics over a shared bottleneck link. Three TCP flows are sent from hosts in the bottom left of the topology to hosts in the bottom right across the link from the NPR to the NetFPGA. Two of the flows share an outgoing queue at the bottleneck link and the third flow has its own queue. The link capacity is set at 500 Mb/s, the shared queue is 1 MB in size, and the non-shared queue is 500 KB in size. The first flow is a long-lived flow, the second is a medium length flow, and the third is a short flow.

The top right of Figure 3.4 shows the throughput seen by each flow and the bottom right shows the queue lengths at the bottleneck. The first flow consumes the entire bottleneck capacity in the absence of other traffic. Once the second flow begins, the two attempt to converge to a fair share of the link, but the third flow begins before they reach it. The NPR is using a typical Weighted Deficit Round Robin scheduler. The user has configured the two queues to receive an equal share of the capacity by setting their scheduling quanta to be the same. The result is that the third flow gets 250 Mb/s of the 500 Mb/s link because it is the only flow in its queue, and the first two share the remaining 250 Mb/s because they are sharing a queue.

```
<?xml version="1.0"?>
<subtype typeName="IPv4Router" version="1">
  <hwType typeName="NetFPGA">
    <resource>NetFPGA.hw</resource>
  </hwType>
  <componentType>router</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
      <default>/users/onl/netfpga/ipv4_router</default>
    </param>
  </init>
  <commands/>
  <monitoring/>
  <ports>
    <fields/>
    <commands/>
    <monitoring/>
    <tables>
      <routeTable title="RouteTable">
        <fields/>
        <commands/>
        <monitoring/>
        <entry name="route" numFields="3" numColumns="2" enable="false">
          <fields>
            <field editable="true">
              <plabel>prefix</plabel>
              <ptype>ipaddress</ptype>
              <default>0.0.0.0</default>
            </field>
            <field editable="true">
              <plabel>mask</plabel>
              <ptype>int</ptype>
              <default>32</default>
            </field>
            <field editable="true">
              <plabel>nexthop</plabel>
              <ptype>nexthop</ptype>
              <default>0</default>
            </field>
```

Listing 3.5: (a) NetFPGA-IPv4Router.shw; IPv4 router specialization for the NetFPGA.

```
45          </fields>
46          <commands>
47            <addCommand opcode="73" numParams="3">
48              <displayLabel>AddRoute</displayLabel>
49              <description>add_route</description>
50              <param editable="true">
51                <plabel>prefix</plabel>
52                <ptype>field</ptype>
53              </param>
54              <param editable="true">
55                <plabel>mask</plabel>
56                <ptype>field</ptype>
57              </param>
58              <param editable="true">
59                <plabel>nexthop</plabel>
60                <ptype>field</ptype>
61              </param>
62            </addCommand>
63            <deleteCommand opcode="75" numParams="3">
64              <displayLabel>DeleteRoute</displayLabel>
65              <description>delete route</description>
66              <param editable="false">
67                <plabel>prefix</plabel>
68                <ptype>field</ptype>
69              </param>
70              <param editable="false">
71                <plabel>mask</plabel>
72                <ptype>field</ptype>
73              </param>
74              <param editable="false">
75                <plabel>nexthop</plabel>
76                <ptype>field</ptype>
77              </param>
78            </deleteCommand>
79            <command opcode="74" numParams="3">
80              <displayLabel>Update Nexthop</displayLabel>
81              <description>update route's next hop</description>
82              <param editable="false">
83                <plabel>prefix</plabel>
84                <ptype>field</ptype>
85              </param>
86              <param editable="false">
87                <plabel>mask</plabel>
88                <ptype>field</ptype>
89              </param>
```

Listing 3.5: (b) NetFPGA-IPv4Router.shw; IPv4 router specialization for the NetFPGA.

```
90              <param editable="true">
91                <plabel>nexthop</plabel>
92                <ptype>field</ptype>
93              </param>
94            </command>
95          </commands>
96          <monitoring/>
97          <display>
98            <column title="prefix/mask" width="120">
99              <fieldName>prefix</fieldName>
100             <symbol>/</symbol>
101             <fieldName>mask</fieldName>
102           </column>
103           <column title="nexthop" width="50">
104             <fieldName>nexthop</fieldName>
105           </column>
106         </display>
107       </entry>
108     </routeTable>
109   </tables>
110 </ports>
111 </subtype>
```

Listing 3.5: (c) NetFPGA-IPv4Router.shw; IPv4 router specialization for the NetFPGA.

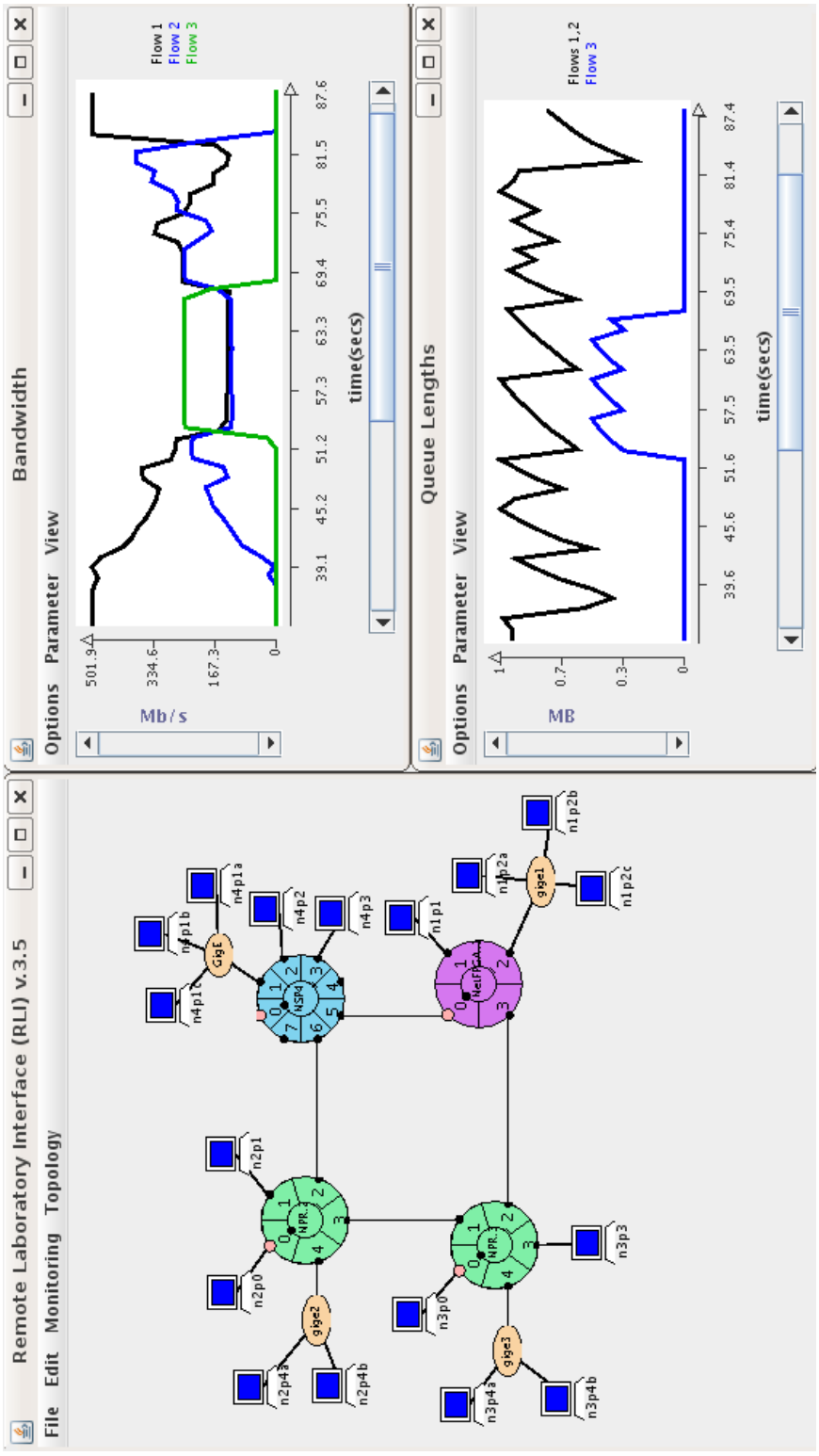

Figure 3.4: An IPv4 network featuring three different types of routers.

3.2.4 NetFPGA Traffic Generator Specialization

The final example specialization for the NetFPGA is a packet generator [20]. NetFPGAs are an excellent platform for traffic generation as they are able to produce line rate traffic on all four ports for packets of any size. This packet generator is trace-based, i.e., it builds and sends packets based on previously recorded data. In this case, packets are specified in the standard PCAP file format used by many network monitoring tools. The specialization description for the NetFPGA packet generator is shown in Listing 3.6.

As the listing shows, this specialization supports two commands. The first is a per-port command that is used to set the packet generation parameters. Each of the four ports is used independently, and there are four parameters needed to configure each port: the packet trace file, the rate to send packets, the number of iterations through the trace file, and the inter-packet delay. Note that the packet rate and inter-packet delay are related and so one of the two must be set while the other must be "-1." The second command is the "SendPackets" command that tells the NetFPGA to begin sending packets based on the per-port configuration commands. The per-port configuration parameters are retained across multiple invocations of the "SendPackets" command, but can be modified at any time or removed by setting the trace file parameter to the empty string.

An example user session utilizing the NetFPGA packet generator specialization is shown in Figure 3.5. For this example, NetFPGA packet generators are used to stress test the NPR capabilities. The topology is given in the top of the figure. There are actually two separate networks in this configuration. The one on the left is used to generate the packet traces needed for the packet generator, and the one on the right is used to perform the actual stress test. To generate the traces, the hosts send packets via normal socket programs and the user runs tcpdump on the nodes to record the packet streams.

The bottom of Figure 3.5 shows the results from one particular stress test. The NPR natively supports IP multicast (details in the next section), and this test is meant to determine if the packet *fanout* has any effect on peak output rate for minimum size packets. The fanout is the replication factor for each incoming packet. Four sets of

```
<?xml version="1.0"?>
<subtype typeName="PacketGenerator" version="1">
  <hwType typeName="NetFPGA">
    <resource>NetFPGA.hw</resource>
  </hwType>
  <componentType>host</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
      <default>/users/onl/netfpga/packet_generator</default>
    </param>
  </init>
  <commands>
    <command opcode="100" numParams="0">
      <displayLabel>SendPackets</displayLabel>
      <description>send packets based on set trace parameters</description>
    </command>
  <monitoring/>
  <ports>
    <fields/>
    <commands>
      <command opcode="101" numParams="4">
        <displayLabel>SetTraceParams</displayLabel>
        <description>set packet trace parameters</description>
        <param editable="true">
          <plabel>trace_file</plabel>
          <ptype>string</ptype>
        </param>
        <param editable="true">
          <plabel>pkt_rate_kbps</plabel>
          <ptype>int</ptype>
          <default>100</default>
        </param>
        <param editable="true">
          <plabel>num_iterations</plabel>
          <ptype>int</ptype>
          <default>1</default>
        </param>
```

Listing 3.6: (a) NetFPGA-PacketGenerator.shw; Packet generation specialization for the NetFPGA.

```
        <param editable="true">
          <plabel>inter_pkt_delay_ns</plabel>
          <ptype>int</ptype>
          <default>-1</default>
        </param>
      </command>
    </commands>
    <monitoring/>
  </ports>
</subtype>
```

Listing 3.6: (b) NetFPGA-PacketGenerator.shw; Packet generation specialization for the NetFPGA.

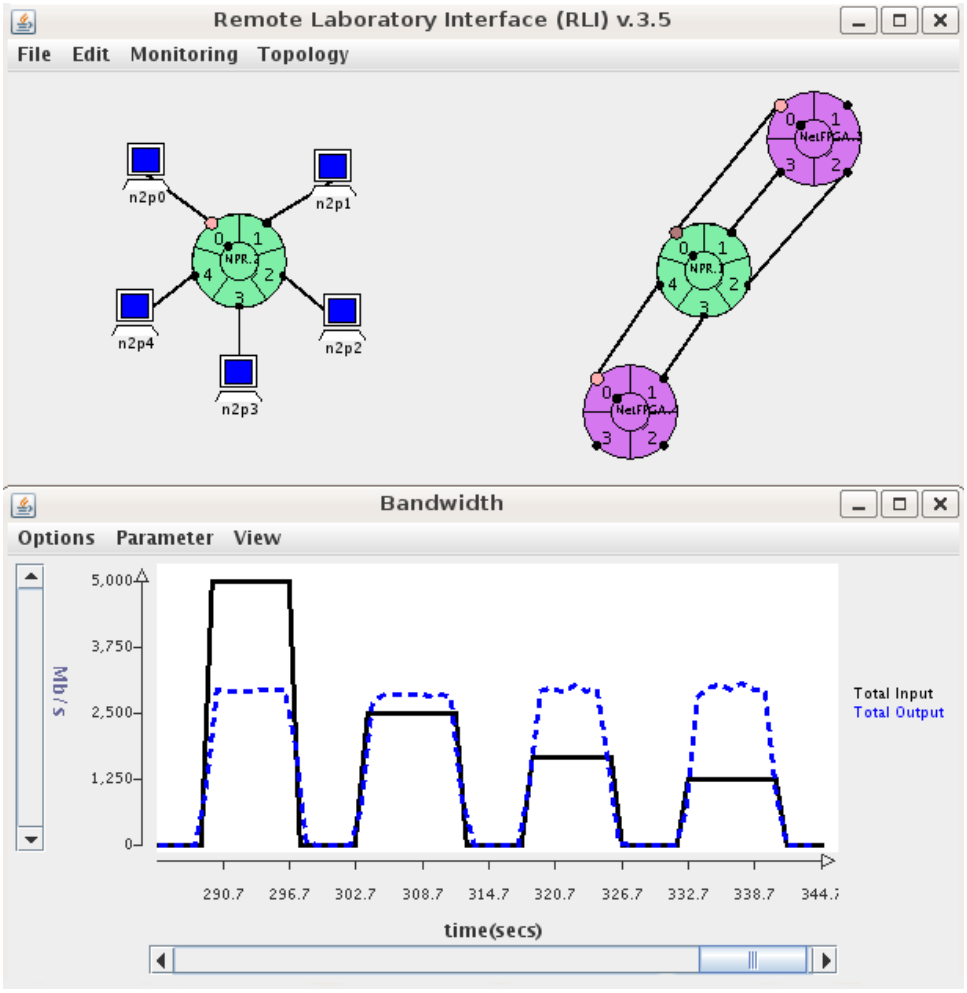

Figure 3.5: Utilizing a NetFPGA to stress test a router.

traffic are sent to the NPR in succession. The first set has a fanout of one, meaning that each input packet is sent out one other port. The second set has a fanout of two, the third has a fanout of three, and the last has a fanout of four. In each case, the total input rate is set so that the total output rate would be 5 Gb/s if the router dropped no packets, and the input and output rates are each shared equally by all 5 ports. The chart shows the aggregate input rate (solid line) and output rate (dashed line) at the router. In each test, the output rate is around 3 Gb/s, showing that the fanout has little effect on the eventual output rate. Other tests confirm that the peak forwarding rate for minimum size packets is around 3 Gb/s for unicast traffic as well.

3.3 IXP

The final type described in this chapter is the Intel IXP 2800 [1]. The IXP 2800, like most network processors, is designed specifically for rapid development of high performance networking applications. The Open Network Laboratory uses one specific IXP-based platform. The Radisys ATCA-7010 [56] is a server blade that uses ATCA-based technology, which is an industry standard for networking components that has broad industry support. ATCA technology is also proving to be a boon to networking research, as it enables the assembly of powerful, yet highly flexible experimental networking platforms. Each Radisys ATCA-7010 board has two IXP 2800s, a shared TCAM, and ten 1 Gb/s data interfaces. In our context, we use the two IXPs as separate five port nodes, assigning five of the data interfaces to each IXP.

The base description for the IXP is shown in Listing 3.7. Once again, the base description is very similar to the previous base descriptions. The IXP, like the NetFPGA, has a default "componentType" of "any" as it could easily be used to fill many different roles in the network. The only other noteworthy difference is that the "clusterOnly" field is set to "true" for the IXP. This is because one physical ATCA board is being treated as two separate IXP nodes and so each pair must be allocated together. Currently, the RLI includes some IXP-specific code to handle this additional requirement.

Before moving on to describe the IPv4 router specialization of the IXP, some additional architectural details about the IXP 2800 are given. In general, the IXP has

```
<?xml version="1.0"?>
<hardware typeName="IXP" version="1">
  <clusterOnly>true</clusterOnly>
  <daemon>3551</daemon>
  <componentType>any</componentType>
  <fields/>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
    </param>
  </init>
  <commands/>
  <monitoring>
    <command opcode="62" numParams="2">
      <displayLabel>UserData</displayLabel>
      <description>user data read from file</description>
      <units>unknown</units>
      <param editable="true">
        <plabel>file</plabel>
        <ptype>string</ptype>
      </param>
      <param editable="true">
        <plabel>field</plabel>
        <ptype>int</ptype>
      </param>
    </command>
    <command opcode="63" numParams="2">
      <displayLabel>UserDataTS</displayLabel>
      <description>user data read from file with time stamp</description>
      <units>unknown</units>
      <param editable="true">
        <plabel>file</plabel>
        <ptype>string</ptype>
      </param>
      <param editable="true">
        <plabel>field</plabel>
        <ptype>int</ptype>
      </param>
```

Listing 3.7: (a) IXP.hw; the base description for the IXP 2800.

```
42      </command>
43    </monitoring>
44    <ports numPorts="5" interfaceType="1G">
45      <fields/>
46      <commands/>
47      <monitoring/>
48    </ports>
49  </hardware>
```

Listing 3.7: (b) IXP.hw; the base description for the IXP 2800.

some unusual architectural features relative to general purpose processors that heavily influence how it is used.

Figure 3.6 is a block diagram showing the major components of the IXP. To enable high performance, while still providing application flexibility, there are 16 multi-threaded MicroEngine (ME) cores that are responsible for the majority of the packet processing in the system. Each ME has eight hardware thread contexts (i.e., eight distinct sets of registers). Only a single thread is active at any given time on one ME, but context switching takes a mere 2-3 clock cycles (about 1.5-2 ns). These threads are the mechanism by which IXP applications deal with the memory latency gap. Indeed, there are no caches in the IXP because caches are not particularly effective for networking applications that exhibit poor locality-of-reference. There is no thread preemption, so a cooperative programming model must be used. Typically, threads pass control from one to the next in a simple round-robin fashion. This is accomplished using hardware signals, with threads commonly yielding control of the processor whenever they need to access memory that is not local to the ME. This round-robin style of packet processing also provides a simple way to ensure that packets are forwarded in the same order they are received. Finally, each ME has a small hardware FIFO connecting it to one other ME, which enables a fast pipelined application structure. These FIFOs are known as next neighbor rings.

The IXP 2800 also comes equipped with 3 DRAM channels and 4 SRAM channels. There is an additional small segment of data memory local to each ME along with a dedicated program store with a capacity of 8K instructions. A small, shared on-chip scratchpad memory is also available. Generally, DRAM is used only for packet buffers. SRAM contains packet meta-data, ring buffers for inter-block communication, and large system tables. In our systems, one of the SRAM channels also supports a

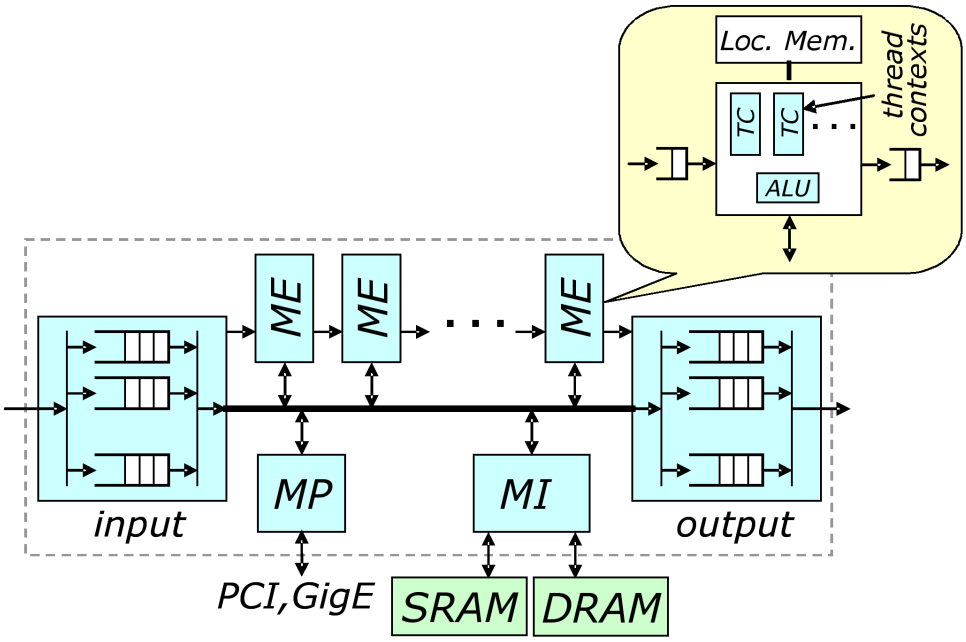

Figure 3.6: Basic block diagram representation of the IXP 2800 [66].

TCAM, which is used for IP route lookup and packet classification. The scratchpad memory is used for smaller ring buffers and tables.

Finally, there is an (ARM-based) XScale Management Processor, labeled "MP" in Figure 3.6, that is used for overall system control and any tasks that are not handled by the rest of the data path. The XScale can run a general-purpose operating system like Linux or a real-time operating system like VxWorks. Libraries exist that provide applications on the XScale direct access to the entire system, including all memory and the MEs.

3.4 Network Processor-based Router Specialization for the IXP

Unfortunately, Network Processors (NPs) like the IXP pose significant challenges to research users. While network equipment vendors can afford to invest significant time and effort into developing NP software, it is more difficult for academic researchers to develop and maintain the necessary expertise. There are several reasons that NPs

are challenging to use. First, it can be difficult to obtain NP-based products, since manufacturers of those products sell primarily to equipment vendors and not to end users. Second, developing software for NPs is challenging because it requires programming for parallel execution, something most networking researchers have limited experience with, and because it requires taking advantage of hardware features that are unfamiliar and somewhat idiosyncratic. Third, there is no established base of existing software on which to build, forcing researchers to largely start from scratch.

The Network Processor-based Router (NPR) was designed, implemented and deployed in order to address these challenges. The NPR is relatively easy to use and can be readily extended through the addition of software *plugins*. All standard router tasks such as route lookup, packet classification, queue management, and traffic monitoring are handled natively, which enables users to focus on new network architectures and services without the need to develop an entire router from the ground up. In addition, a plugin environment is provided with significant processing and memory resources, and an API that implements common functions needed by most users. This makes it possible for users to develop plugins for non-trivial network experiments without having to write a substantial amount of new code.

This section will describe the NPR in detail. The NPR specialization description is too large to include directly here, but it is given in Appendix B.

3.4.1 Data Plane

The software organization and data flow for the NPR are shown in Figure 3.7. Note the allocation of MEs to different software components. The main data flow proceeds in a pipelined fashion starting with the *Receive block* (Rx) and ending with the *Transmit block* (Tx). Packets received from the external links are transferred into DRAM packet buffers by Rx. Rx allocates a new buffer for each incoming packet and passes a pointer to that packet to the next block in the pipeline, along with some packet meta-data. In order to best overlap computation with high latency DRAM operations, Rx processing is broken up into two stages with each stage placed on a separate ME. Packets flow from the first stage to the second over the next neighbor ring between the MEs.

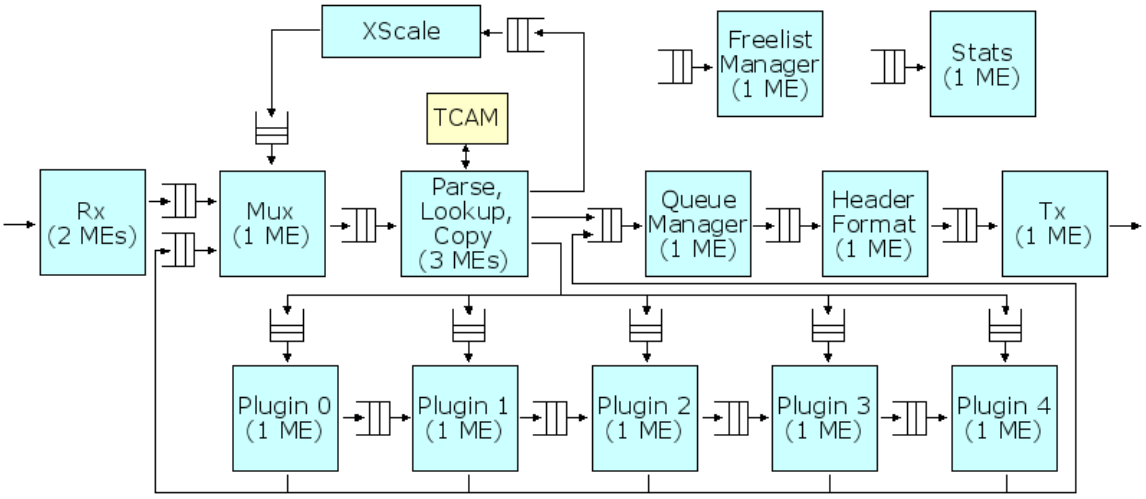

Figure 3.7: Data plane of the NPR.

Note that, in general, the information passed between blocks in the diagram consists of packet references and selected pieces of header information, not packet data. Thus, packets are not copied as they move (logically) from block to block. Each block can access the packet in DRAM using its buffer pointer, but since DRAM accesses are relatively expensive, the system attempts to minimize such accesses. Also note that most blocks operate with all eight threads running in the standard round-robin fashion.

The *Multiplexer block* (Mux) serves two purposes. Each packet buffer in DRAM has an associated 32B entry in SRAM that stores some commonly needed information about the packet, such as the packet length. Mux initializes this meta-data for packets coming from Rx. Its second function is to multiplex packets coming from blocks other than Rx back into the main pipeline. This includes packets coming from the XScale and packets coming from plugins. A simple user-configurable priority is used to determine how to process packets from the different sources.

The *Parse, Lookup, and Copy block* (PLC) is the heart of the router. Here, packet headers are inspected to form a lookup key that is used to find matching routes or filters in the TCAM. Routes and filters are added by the user, and will be discussed further in Section 3.4.3. Based on the result of the TCAM lookup, PLC takes one of five actions. First, the packet can be sent out towards the external links via the *Queue Manager block* (QM). Second, the packet can be sent to the XScale if it has

some special processing needs not handled by the ME pipeline. Third, the packet can be sent to a plugin ME (hosting user code). Plugins will be discussed fully in Section 3.4.3, but, as can be seen in Figure 3.7, plugins will be able to forward packets to many other blocks including the QM, Mux, and other plugins. Fourth, the packet can be dropped. Finally, multiple references to the same packet can be generated and sent to distinct destinations. For example, one reference may be sent to a plugin and another directly to the QM. In fact, this mechanism allows the base router to support IP multicast (among other things). Reference counts are kept with the packet meta-data in SRAM to ensure that the packet resources are not reclaimed prematurely. Note that this does not allow different "copies" to have different packet data because there is never more than one actual copy of the packet in the system.

Before moving on, it is worth discussing the design choices made for PLC. Three MEs all run the entire PLC code block with all 24 threads operating in a round-robin fashion. One alternative would be to break up the processing such that Parse is implemented on one ME, Lookup on a second, and Copy on a third. Our experience shows that the integrated approach chosen here yields higher performance. This is primarily due to the nature of the operations in PLC. To form the lookup key, Parse alternates between computation and high latency DRAM reads of packet headers, and Lookup spends most of its time waiting on TCAM responses. On the other hand, Copy is computation-bound due to the potentially complex route and filter results that must be interpreted. Combining all three blocks together provides enough computation for each thread to adequately cover the many memory operations.

Continuing down the main router pipeline, the QM places incoming packets into one of 8K per-interface queues. A weighted deficit round robin scheduler (WDRR) is used to pull packets back out of these queues to send down the pipeline. In fact, there is one scheduler for each external interface, servicing all the queues associated with that interface. Each queue has a configurable WDRR quantum and a configurable discard threshold. When the number of bytes in the queue exceeds the discard threshold, newly arriving packets for that queue are dropped. The QM has been carefully designed to get the best possible performance using a single ME. The design uses six threads. One handles all enqueue operations, and each of the remaining five implements the dequeue operations for one outgoing interface. This decouples the two basic tasks and enables the QM to achieve high throughput.

Following the QM is the *Header Format block* (HF) that prepares the outgoing Ethernet header information for each packet. It is responsible for ensuring that multiple copies of a packet (that have potentially different Ethernet source addresses) are handled correctly. Finally, the *Transmit block* (Tx) transfers each packet from the DRAM buffer, sends it to the proper external link, and deallocates the buffer.

There are two additional blocks that are used by all the other blocks in the router. The first is the *Freelist Manager block* (FM). Whenever a packet anywhere in the system is dropped, or when Tx has transmitted a packet, the packet reference is sent to the FM. The FM then reclaims the resources associated with that packet (the DRAM buffer and the SRAM meta-data) and makes them available for re-allocation. The *Statistics block* (Stats) keeps track of various counters through-out the system. There are a total of 64K counters that can be updated as packets progress through the router. Other blocks in the system issue counter updates by writing a single word of data to the Stats ring buffer, which includes the counter to be updated and increment to be added. For example, there are per-port receive and transmit counters that are updated whenever packets are successfully received or transmitted, respectively. There are also counters for each route or filter entry that are updated both before and after matching packets are queued in the QM. This provides a fine-grained view of packet flow in the router. The counters all ultimately reside in SRAM, but there are 192 counters that are also cached locally in the Stats ME. One thread periodically updates the SRAM counterparts to these counters while the other threads all process update requests from the ring buffer.

The final block in the diagram is the XScale. The primary purpose of the XScale is to control the operation of the data plane, as discussed in detail in the next section. However, the XScale also plays a small role in the data plane. Specifically, it handles any IP control packets and other exceptional packets. This includes all ICMP and ARP handling as defined by the standard router RFCs. All messages generated by the XScale are sent back through Mux to PLC, allowing users to add filters to redirect these packets to plugins for special processing, should they desire to do so.

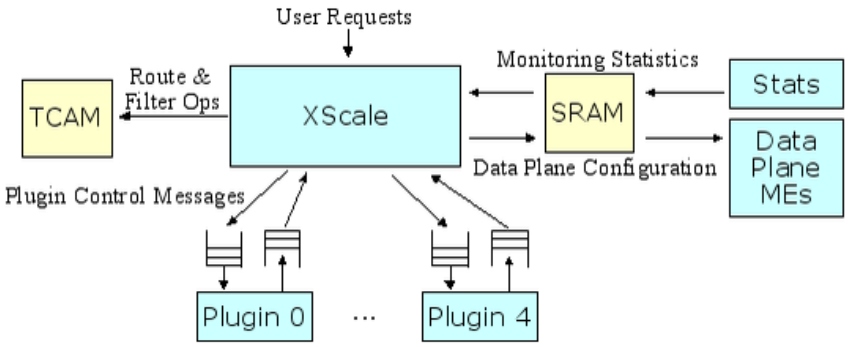

Figure 3.8: Control plane of the NPR.

3.4.2 Control Plane

The XScale's main function is to control the entire router and act as the intermediary between the user and the data plane. This is accomplished by a user-space control daemon running under Linux on the XScale. Libraries provided by Intel are used to manage the MEs and the memory in the router, and a library provided by the TCAM vendor contains an API for interacting with the TCAM. Figure 3.8 summarizes the most important roles of the XScale daemon. Messages come from the user to request certain changes to the system configuration. Typically the first such request is to start the router, which involves loading all the base router code (i.e., everything except plugins) onto the MEs and enabling the threads to run. Once the data path has been loaded successfully, there are several types of control operations that can be invoked.

The first of these involves configuration of routes and filters, which are discussed in detail in Section 3.4.3. The XScale also supports run-time configuration of some data plane blocks, by writing to pre-defined control segments in SRAM. For example, queue thresholds and quanta used by the QM can be dynamically modified by the user in this way.

The XScale also provides mechanisms to monitor the state of the router. All of the counters in the system are kept in SRAM, which allows the XScale to simply read the memory where a certain counter is stored to obtain the current value. These values can be sampled periodically in order to drive real-time displays of packet rates, queue

lengths, drop counters, etc. The system can support dozens of these ongoing requests concurrently.

All plugin operations are also handled by the XScale. In particular, this involves adding and removing user-developed plugin code on the plugin MEs and passing control messages from the user to the plugins after they are loaded. The control messages are forwarded by the XScale through per-plugin rings, and replies are returned in a similar way. The XScale is oblivious to the content of these control messages, allowing users to define whatever format and content is most appropriate for their applications. Plugins can also effect changes to the router by sending requests through these rings to the XScale. Once a plugin is loaded, users can add filters to direct specific packet flows to the plugin.

3.4.3 Programmability

Now that the basics of the NPR have been covered, we turn our attention to the progammable components of the router. There are two primary facets of the overall programmability of the NPR: plugins and filters. Together they provide users a rich selection of options with which to customize processing and packet flow in the NPR.

Plugin Framework

Recall from Figure 3.7 that five MEs are used to host plugins. NPR users are free to load any combination of code blocks onto these MEs. In addition to the five MEs, there are five ring buffers leading from PLC to the plugins which can be used in any combination with the MEs. For example, each plugin may pull packets from a separate ring (this is the default behavior) or any subset of plugins may pull from the same ring. The ring buffers, then, are a level of indirection that adds versatility to potential plugin architectures. The NPR also sets aside 4KB of the scratchpad memory and 5MB of SRAM exclusively for plugin use.

The actual processing done by any particular plugin is entirely up to the plugin developer. Plugins are written in MicroEngine C, which is the standard C-like language provided by Intel for use on MEs. The most important differences between MicroEngine C and ANSI C are dictated by the IXP architecture. First, there is no dynamic memory allocation or use because there is no OS or other entity to manage the memory. Second, all program variables and tables must be explicitly declared to reside in a particular type of memory (registers, ME local memory, scratchpad, SRAM, DRAM) as there is no caching. Finally, there is no stack and hence no recursion. Also recall from the IXP review that the eight hardware contexts share control explicitly (no preemption).

To help users who are unfamiliar with this programming environment, we have developed a framework that lowers the entry barrier for writing simple to moderately complex plugins. Our framework consists of a basic plugin structure that handles tasks common to most plugins and a plugin API that provides many functions that are useful for packet processing.

In the basic plugin structure, there are three different types of tasks to which the eight threads are statically assigned at plugin compile time. The first of these tasks deals with packet handling. The framework takes care of pulling packets from the incoming ring buffer and then calls a user supplied function to do the actual plugin processing. When the function returns, the packet is pushed into the outgoing ring. As can be seen in Figure 3.7, the packet can be sent back to MUX that results in the packet being matched against routes and filters in the TCAM a second time. This is useful if something in the packet, such as the destination IP address, has changed and the packet might need to be re-routed. Alternatively, the plugin can send the packet directly to the QM so that it will be sent out to the external links. Packets can also be redirected to the next plugin ME via the next neighbor rings. In fact, although it is not shown in the figure to avoid confusion, plugins even have the ability to send packets to any other plugin ME by writing directly to the five ring buffers leading from PLC to the plugins.

The second type of task is the periodic task. Some plugins may need to do processing that is not dictated purely by packet arrivals. In such cases, plugin developers can assign threads to the periodic task that has the thread sleep for a configurable time

and then call another user provided function to do the periodic processing. The last type of task is the control task, first mentioned in Section 3.4.2. This provides a mechanism for plugins to receive and respond to control messages from the RLI. As an example, we have a plugin that delays packets by N ms, where N can be set via a control message. These messages and their responses go through per-plugin control ring buffers as shown in Figure 3.8. These rings are also used for plugins to request modifications to the system outside the plugin MEs. Such requests are processed by the XScale and allow plugins to modify queue parameters, add or remove routes and filters, and even add or remove plugin code from other plugin MEs.

To support plugin developers, we provide a plugin API. The API consists of helper functions for common packet processing steps as well as functions that hide some of the complexity of interacting with packets and packet meta-data.

Filters

In order to actually get packets to plugins, filters are installed to direct packet flows to specific destinations in the router. More generally, filters are used to modify default behavior of the router by superseding standard IP routing. As mentioned in Section 3.4.1, filters and routes are stored in the TCAM. Routes are simpler and used only for standard IP routing. That is, the packet's destination IP address is compared to the route database and the longest matching address prefix is returned. The result also contains the external interface that the packet should be forwarded on. Each NPR supports 16K routes.

Filters are more general than routes and include more fields in the lookup key and more options for the action to be taken. Figure 3.9 shows a screenshot of the dialog box provided by the RLI for adding filters. In general, the fields in the top half of the window constitute the lookup key and those in the bottom half describe what should happen to any matching packets. Each key field can be specified as a particular value or as "*," which means that any value for that field should match. An expanded version of the standard IP 5-tuple forms the core of the lookup key, including source and destination IP address ranges, source and destination transport protocol ports, and the IP protocol. For TCP packets, the TCP state flags are also part of the key

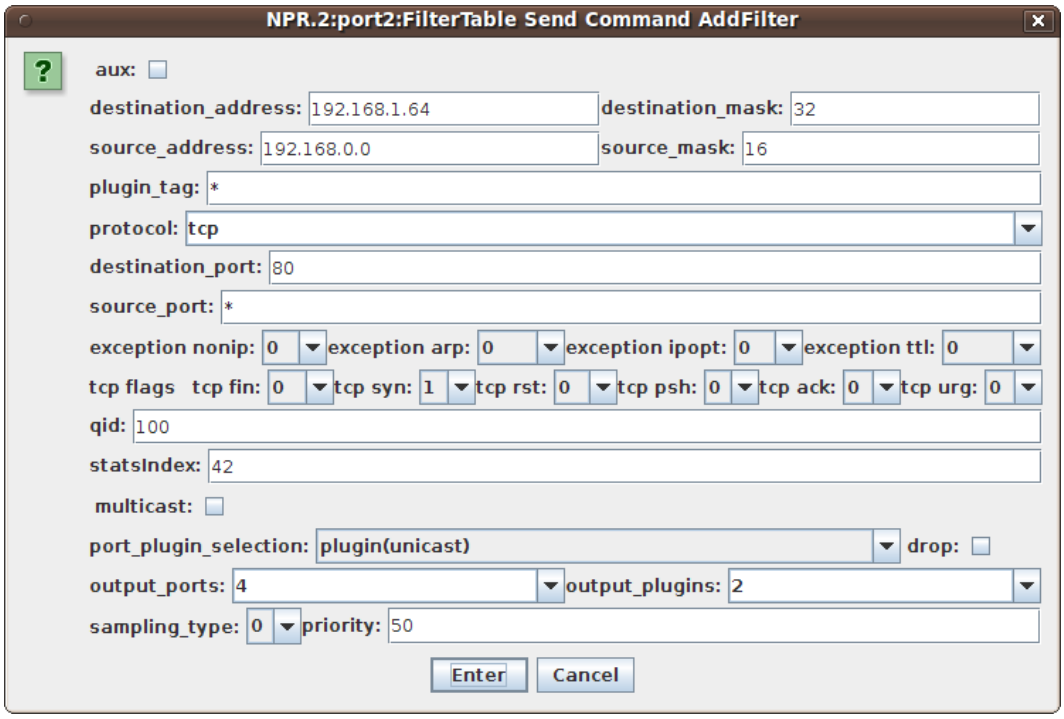

Figure 3.9: Adding a filter in the RLI.

allowing filters to match particular parts of TCP flows. The *plugin_tag* is a 5 bit field that plugins can modify to force different matches to occur on any subsequent passes the packet takes through PLC. Finally, there are the *exception_bits* that allow exceptional traffic, such as packets with a Time-to-Live of 0, to be matched and potentially handled in some way other than the default. In this particular example, the beginning of any HTTP flow from hosts in the 192.168.0.0/16 subnet going to 192.168.1.64 will be matched. Note that the TCP flags indicate only TCP SYN packets (and not SYN-ACKs) will match. This works by setting the *tcpflags_mask* to be all ones, i.e., all flags must match exactly. Then, only the second bit (the SYN bit) is set in the actual *tcpflags* field.

The rest of the fields determine exactly what happens to packets that match the filter. The most important fields are *port_plugin_selection*, *output_ports*, and *output_plugins* because they determine whether or not the matching packets will go directly to the QM or to a plugin. In the figure, the filter is configured to send the packets to plugin 2. The *output_ports* field is part of the data that is passed to the plugin as well, so that the plugin knows where to send the packet next (if it decides to forward the

packet). Note that the *output_plugins* field actually refers to the ring buffer leading by default to that plugin ME, but any plugin is capable of reading from any of the five rings, so plugin developers are free to configure plugins to process packets from the ring buffers in other ways as well. The *multicast* field can be set if multiple copies of the packet are desired. In that case, any combination of ports and plugins can be set in the *output_ports* and *output_plugins* fields, and copies will be sent by PLC to each of the specified destinations. The *qid* determines which of the 8K per-interface queues the packet enters when it reaches the QM. In the event of multiple copies, each copy will go into the same numbered queue for whichever interface it is destined. Users can also specify that all matching packets should be dropped by selecting the *drop* field.

There are actually two different filter types in the NPR, differentiated by the selection of the *aux* field. If the field is not set, the filter is called a *primary* filter, and if it is, an *auxiliary* filter. The lookup key fields and actions all have the same meaning for either type, but auxiliary filters cause an additional reference to the matching packet to be created by PLC and sent to the destination contained in the auxiliary filter. This means that auxiliary filters represent a separate set of potential matches. On the other hand, primary filters are logically in the same set of potential matches as routes. This is where the *priority* field in the filter comes into play. All routes are assigned the same priority while each filter has its own priority. When a packet matches multiple primary filters, the highest priority filter is considered to be the matching one, unless the route priority is higher. In that case, the matching route is used to determine how the packet is forwarded. For auxiliary filters, the highest priority auxiliary filter is considered to be a match. Although matching packets against filters with priorities can potentially be fairly complex, the TCAM allows us to lay out all routes and filters in such a way that higher priority entries (and longer prefixes for routes) come first in the TCAM tables and are thus the first match returned when the TCAM is queried. The end result is that a single packet can match one primary filter or route, and one auxiliary filter. This can be quite useful if the user wishes to have passive plugins that monitor certain packet streams without disturbing the normal packet flow. Each NPR supports 32K primary filters and 16K auxiliary filters.

3.4.4 Example Plugins

To provide a more concrete view of the capabilities of the NPR plugin environment, we now describe three example plugins that have been written and tested in the router.

Network Statistics

The first example is a simple plugin that keeps a count of how many packets have arrived with different IP protocols. The code written by the developer is shown in Listing 3.8 and consists mostly of API calls to read the IPv4 header from the packet in DRAM. First, the packet reference is filled in from the input ring data by calling `api_get_pkt_ref()`. The packet itself resides in a 2KB DRAM buffer with the beginning of the packet header at some offset into that buffer (to accommodate packets that may increase in size). The offset is part of the meta-data for that packet, so the second step is to read the meta-data using `api_read_meta_data()`. Once we have the offset, `api_get_ip_hdr_ptr()` is called to calculate the address of the IP header. `api_read_ip_hdr()` reads the header into a local struct, which grants easy access to the header fields. Finally, based on the IP protocol one of four different plugin-specific counters is incremented with `api_increment_counter()`. These counters can be monitored easily in the RLI so that the user can see, in real time, how many packets of each type are passing through the plugin. The plugin does not explicitly decide where packets should go next and so by default all packets will be sent to the QM after leaving the plugin. This plugin has no need for periodic tasks or for control messages from the RLI so all eight threads are devoted to handling packets as they arrive.

Network Support for Distributed Games

Our next example is a system that provides network services in support of highly interactive distributed games. The network provides support for a distributed collection of game servers that must share state information describing the current status of various objects in the game world (e.g., user avatars, missiles, health packs, etc).

```
1  api_pkt_ref_t packetRef; // packet reference
2  api_meta_data_t metaData; // local copy of meta-data
3  unsigned int ipHdrPtr; // pointer to IP header (DRAM)
4  api_ip_hdr_t ipHdr;    // local copy of IP header
5
6  api_get_pkt_ref(&packetRef);
7  api_read_meta_data(packetRef, &metaData);
8
9  ipHdrPtr = api_get_ip_hdr_ptr(packetRef,
10                                metaData.offset);
11 api_read_ip_hdr(ipHdrPtr, &ipHdr);
12
13 switch(ipHdr.ip_proto)
14 {
15   case PROTO_ICMP: api_increment_counter(0); break;
16   case PROTO_TCP : api_increment_counter(1); break;
17   case PROTO_UDP : api_increment_counter(2); break;
18   default        : api_increment_counter(3); break;
19 }
```

Listing 3.8: nstats.c; code for the network statistics plugin.

State update packets are distributed using a form of *overlay multicast*. These updates are labeled with the game world *region* where the associated object is located and each region is associated with a separate multicast data stream. Servers can *subscribe* to different regions that are of interest to them, allowing them to control which state updates they receive.

Figure 3.10 shows an example ONL session for the distributed game application. This configuration uses 12 routers and 36 end systems acting as game servers, each supporting up to 10 players. The routers in this system host two distinct plugins that implement the region-based multicast. While it would have been possible to implement this application using the built-in IP multicast support, the use of application-level multicast frees the system from constraints on the availability of IP addresses, and enables a very lightweight protocol for updating subscriptions, making the system very responsive to player activity.

The first of our two plugins is a *Multicast Forwarder* that forwards the state update packets sent by the game servers, and the second is a *Subscription Processor*. The subscription state for each region in the game world is a bit vector specifying the subset of the five outgoing links that packets with a given region label should be

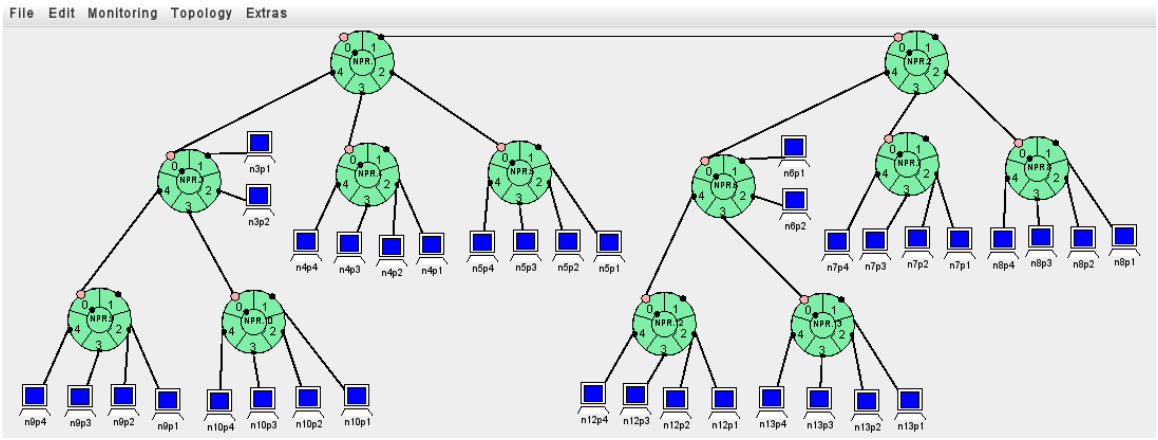

Figure 3.10: Example topology for a distributed game application.

forwarded to. These bit vectors are stored as a one-dimensional array in SRAM, which is shared by the two plugins. One megabyte of SRAM has been allocated to these multicast bit vectors, allowing up to one million regions in the game world.

The subscription processor runs on one ME and handles subscription messages from all router interfaces. A TCAM filter is configured for each of the router's input ports, directing all subscription packets to the subscription processor's input ring. Each of these messages contains one or more records with each record containing a join or a leave request for one region. The subscription processor reads each of these records from the packet in DRAM and updates the corresponding state in the subscription bit vector. All eight threads are used to process subscription packets.

Four of the MEs are used to host multicast forwarders. These MEs share a single input ring and process different packets in parallel. Altogether, 32 distinct packets can be processed concurrently (using the eight hardware thread contexts in each ME). A TCAM filter is configured for every port, to direct all state update traffic to the shared input ring. To process a packet, the multicast forwarder reads the packet header to determine which multicast stream the packet belongs to and then reads the subscription bit vector for that region to determine which ports the packet should be forwarded to. It then replicates the packet reference information as needed, and forwards these packet references to the appropriate queues in the QM. Note that the packets themselves are never copied.

Regular Expression Matching

The last and most complex example is a set of plugins that are used for high speed regular expression matching of packet data using Deterministic Finite Automata (DFAs). These could be used for any application based on deep packet inspection, such as network intrusion detection and content-based routing. The basic operation involves following state transitions in a DFA for every input character received, where one DFA encompasses many regular expressions. This topic has been studied extensively and we take advantage of many state of the art techniques in our plugins.

Regular expression matching is a memory intensive application both in terms of memory and bandwidth. To reduce the space requirements, we utilize both default transition compression and alphabet reduction [10]. This allows us to fit more regular expressions into the memory available to plugins. Another technique involves pattern partitioning of the set of regular expressions to form multiple DFAs [78]. This can reduce the overall space needs, but increases the memory bandwidth since all of the DFAs have to be traversed for each input character. Fortunately, the hardware threads available on the plugin MEs provide the necessary parallelism to support this requirement.

As in the previous example, there are two types of plugins. The *Packet Loader* prepares packets for the *Matching Engines* that actually run the DFAs. An auxiliary filter directs a copy of every packet to the Packet Loader. This ensures that even if our plugins fall behind in packet processing and potentially drop packets the actual packet stream will remain unaffected. The Packet Loader reads the packet header and passes the packet payload to the first Matching Engine through the next neighbor ring between them.

Up to four Matching Engines can be added to run on the remaining MEs depending on how many threads are needed. Figure 3.11 shows the design of the Matching Engine. As discussed above, the regular expressions are partitioned to produce a set of DFAs and each DFA is then run exclusively by one thread on one ME. One thread on each Matching Engine is used for synchronization, so up to seven threads can be processing DFAs. There is a one-to-one mapping of matching threads and DFAs, meaning that each matching thread always processes packet data against one

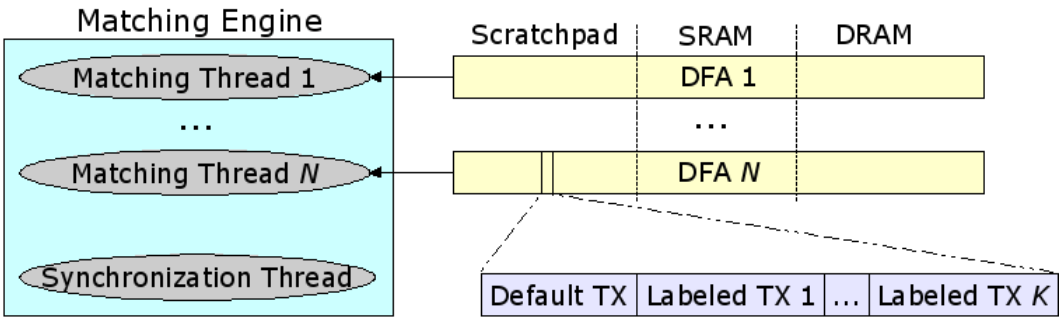

Figure 3.11: Design of the Regular Expression Matching Engine.

particular DFA. Each packet is processed against all DFAs in order to determine if there are any matches.

The synchronization thread has two tasks. The first is to read data from the incoming next neighbor ring so that the local matching threads can access it, as well as passing the data to the next Matching Engine if there is one. The thread's main task is to handle synchronization of packet data across the matching threads. Synchronization is implemented through the use of a circular buffer stored in shared local memory. The synchronization thread writes data to the buffer so that the matching threads can read it. Each matching thread can then proceed through the buffer at its own pace, and the synchronization thread ensures that data that has been processed by all the threads is flushed.

Each matching thread invokes its DFA on the input data one character at a time. Alphabet translation is first performed on the input character (the translation table is stored in local memory). The next state is then found by traversing the DFA, and processing continues with the next character. As is shown in Figure 3.11, the DFAs are stored hierarchically in memory with some state information in the scratchpad memory, some in SRAM, and some in DRAM. The memory layout is generated ahead of time so that states more likely to be reached are stored in faster memory. The figure also shows an example of the actual state information. Recall that we use the default transition method noted above. Each state contains a small number of labeled transitions and a default transition. If none of the labeled transitions are for the current input character, the default transition is taken. The operational details are given in [10].

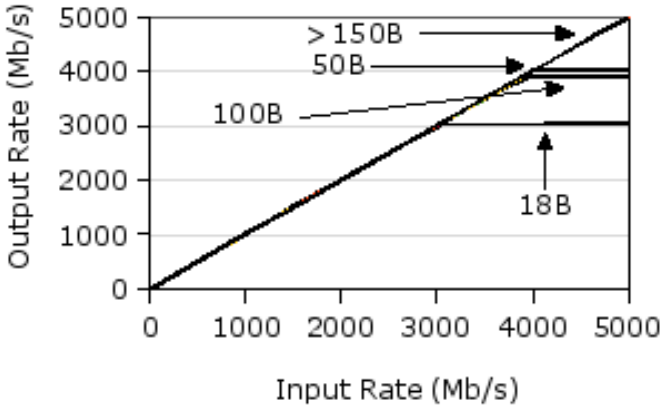

Figure 3.12: Unicast throughput results for the NPR.

3.4.5 Performance

We include here a brief performance evaluation of the NPR as a baseline reference for plugin developers. UDP flows consisting of equally sized packets were directed to the router for aggregate input rates of up to 5 Gb/s, with input and output bandwidth split equally among the five interfaces. Figure 3.12 shows the forwarding rate when packets proceed directly through the main pipeline with no plugins installed. The different curves represent the output rate for flows of different packet sizes, where the labels indicate the UDP payload size of the packets. Note that the input and output rates are reported relative to the external interfaces. As such, an input rate of 5000 Mb/s means that all five input links are completely saturated.

For packets with UDP payloads larger than 150 bytes, the NPR is able to keep up with any input rate. Streams of smaller packets cause the router to drop packets at high input rates. For 100 and 50 byte packets, the output rate levels off near 4 Gb/s. It is interesting to note that the output rate is higher in the 50 byte case than in the 100 byte case. This apparent anomaly is a product of the way the Rx and Tx blocks interact with the external interfaces. Packets actually arrive in fixed-size cells that Rx reassembles into full packets. A UDP payload size of 100 bytes forces incoming packets to take two of these cells while 50 byte UDP payloads can fit in a single cell. This means that Rx has significantly more work to do in the former case, which results in a lower overall output rate. As packets continue to decrease in size, the

peak output rate continues to decrease. For minimum size Ethernet frames (UDP payload of 18 bytes), the output rate is around 3 Gb/s.

We performed the same evaluation with the entire input stream directed through a single *null* plugin. The null plugin uses the basic plugin structure described in Section 3.4.3 with an empty function to handle packets. The packets are then forwarded to the QM. The results were identical, which means that a single plugin is capable of handling every packet that the router can forward.

Native multicast performance was also evaluated, as shown in Figure 3.5, and described in Section 3.2.4.

The NPR was also evaluated with the distributed gaming plugins installed. Recall from Section 3.4.4 that the multicast forwarder plugins are responsible for generating multiple references to incoming packets based on the current subscriptions for that packet's multicast group. In the NPR, the packet replication factor, or *fan-out*, can be between 0 and 4 as there are 5 interfaces and packets are never sent out on the interface on which they arrived. To test the forwarding capability of the multicast plugins, UDP streams with UDP payloads of 150 bytes (a typical size for incremental game state updates) were directed through the plugins. As with the baseline evaluation, the input and output bandwidth was split equally among the five interfaces. Note that as the fan-out increases the input packet rate needed to produce the same output rate decreases. The peak output rates for fan-outs of 1, 2, 3, and 4 are, respectively, 4 Gb/s, 4.5 Gb/s, 4 Gb/s, and 3.6 Gb/s. To understand this result, consider how the workload changes as the fan-out changes. For low fan-outs the plugins have to process more packets per second, but the per-packet work is lower. On the other hand, when the packet rate is lower the per-packet work is significantly higher. This leads to optimal performance when the fan-out is 2, given the characteristics of the router.

To explore the bottlenecks in our system, we used Intel's cycle-accurate simulation environment. Unsurprisingly, no single block is responsible for not keeping up with line rate for small packets. As mentioned above, Rx has limitations for some size packets. For minimum size packets, both the QM and PLC blocks peak between 3 and 3.5 Gb/s. We could enable the main router path to perform better by using more MEs in those cases, but we believe that their use as plugin MEs is ultimately more

beneficial. The traffic conditions that cause the router to perform sub-optimally are also unlikely to occur for extended periods. Under realistic traffic patterns the router is able to keep up with line rate.

We also monitored packet latency during the above evaluation by periodically sending ping packets through the router and recording the round-trip time. When the router is otherwise idle, the average RTT is around 100 μs with a standard deviation of around 2 μs. Note that much of that time is accounted for at the end hosts. The one-way, edge-to-edge latency of the router is no more than 20 μs. As expected due to the IXP architecture, these values do not change significantly with a change in router load. Indeed, even under the heaviest load for minimum-sized packets the average RTT doesn't change and the standard deviation only increases to around 4 μs for packets that are not dropped.

3.4.6 Related Work

Router plugin architectures are certainly not a new idea [21]. Indeed, the NSP [15], described in Section 2.5.1, uses a general purpose processor on each port of a hardware router in order to run user plugins. There have also been other extensible routers based on network processors. For example, [63] describes a router that uses the IXP 1200 network processor for fast-path processing in a standard PC, and [66] describes an ATCA-based platform that allows PlanetLab [55] applications to achieve significantly better performance by incorporating IXP 2850 network processors as fast-path elements and line cards. In the traditional software router space, Click [38] takes a modular approach to extensibility by providing a framework for connecting various forwarding and routing modules together in an arbitrary fashion. XORP [33] is similar in nature to Click, but it is focused on modular routing stack components. Our work is based in part on all of these previous efforts, but aims to fill a different role. The NPR provides users direct access to a significant portion of the router resources, a software base on which to build, and the ability to have router code that performs substantially better than on a standard PC. Moreover, our routers are already deployed in the Open Network Laboratory, which allows anyone to experiment with them in a sandboxed environment.

There is another body of related work that aims specifically to ease software development for NPs. One example is NP-Click [61], which provides a programming model that abstracts away many of the architectural aspects that make programming difficult. Shangri-La [14] is another example where a compiler and thin run-time environment are used to support higher-level programming languages. These systems employ many complex optimizations to bring application performance to similar levels achieved by hand-written assembly code. These approaches are complimentary to the NPR plugin framework. Ultimately, the framework provides direct low-level access to the plugin MEs that allows such solutions to be run directly as any other NPR plugins. Of course, there has been no ongoing support for NP-Click or Shangri-La, whereas the NPR is available for anyone to use.

3.4.7 NPR Conclusions

The NPR has been used in a number of networking and architecture courses. For example, a local advanced architecture class focused on multi-core processors utilized the NPRs for their final projects. There were ten groups and all ten of the projects were completed successfully. The projects ranged from TCP stream reassembly to complex network monitoring to network address translation. Feedback from that class and others has been incorporated into the NPR and particularly into the plugin API. Based on our experience so far, we believe that the NPR is an excellent platform for network study and experimentation. Our plugin framework provides a flexible structure and API that helps users unfamiliar with similar systems to build expertise without hindering experts.

Chapter 4

Testbed Scheduling

Recall from Section 2.4 that most testbeds, including the new framework described in Chapter 2, support multiple concurrent user sessions that are guaranteed to be isolated from one another. This is accomplished with a collection of switches and routers that indirectly connect all of the resources in the testbed and an associated testbed *scheduler*.

The scheduler takes a virtual network request and attempts to find a mapping from the virtual network onto the available physical resources. If a mapping is found, the physical resources in that mapping are allocated to the user and the physical network is configured (typically with VLANs) to emulate the virtual topology. The scheduler must ensure that each physical node is mapped to only one virtual node across all virtual networks. Moreover, there must be enough capacity in the infrastructure switches and routers to support every virtual link without causing interference with other virtual links. Any mapping that meets these requirements could be returned by the scheduler. There are two testbed design choices that affect the associated testbed scheduler.

One of these design choices is how the resources are allocated to users. That is, the resources are either given on-demand or reserved in advance. In the former case, the scheduler looks strictly at the physical resources that are not in use by any other experiment, as in standard admission control. This is how most emulation testbeds operate. The latter case is somewhat more complicated. The scheduler must keep a time line of *reservations* that determines what resources are available at any given time. In addition to the virtual network to be emulated, user requests include a period of time when that virtual network should be active. When a new request is made, all

previously accepted reservations that overlap are considered. Any reservations whose start time has not come (i.e., are not yet active) could potentially be remapped to a new set of physical resources. Clearly, maintaining a schedule of network mappings is a generalization of pure on-demand admission control.

The second testbed design choice is whether or not the testbed nodes are *typed*. In most testbeds, homogeneous PCs are the only user-allocatable resource, meaning that all physical nodes can be treated more or less equally. Other testbeds support heterogeneous resources. This might include many different PC configurations or other networking technologies altogether, such as programmable routers and reconfigurable hardware. In this case, every node is assigned a type that reflects these diverse resource possibilities. Naturally, finding mappings with typed nodes is a more general problem than with non-typed nodes, which results in more complicated schedulers.

In order to support the needs of the general testbed framework, it is necessary to build a testbed scheduler that can operate under the more general design decisions. This chapter will therefore focus on designing schedulers that support both reservations and typed nodes.

Scheduling virtual networks in testbeds is a variant of the general network embedding problem, which is known to be NP-hard. As such, this work will explore heuristic scheduling approaches. In particular, this chapter will present a new class of schedulers that use Mixed Integer Programs (MIPs) to optimally solve key subproblems by incorporating knowledge of the physical network topology of the testbed. These new schedulers are evaluated and results presented for response times to new requests and for the probability of rejecting requests. There are a large number of factors that affect the performance, including the size and shape of the virtual and physical topologies, reservation durations, and user flexibility. This work also provides a new way to characterize the work load for testbed schedulers to assist in understanding the limitations and bottlenecks of different approaches.

Note that this work assumes every testbed node is assigned to at most one user at any given time. In other words, allocation of nodes to users is all or nothing. This is necessary when dealing with many networking technologies that are not easily virtualized or shared by multiple concurrent users. However, it would be relatively

easy to extend this work to incorporate node virtualization for resource types that support it.

4.1 Related Work

Emulab [69] provides access to a large number of heterogeneous PCs that are used to emulate many different types of networks. The Emulab scheduler, *assign*, is based on simulated annealing [58]. It has support for typed nodes in order to take advantage of the different PC classes available in the testbed, but does not support future resource reservations. Emulab does, however, allow users to put their rejected experiment requests into a scheduling queue that *assign* periodically reevaluates. In this case, the scheduler is still only performing admission control. Recent work [59] suggests that Emulab is working on support for a more general resource reservation model.

There is a related embedding problem in overlay testbeds, such as PlanetLab [55]. PlanetLab nodes are PCs connected directly to the Internet, and users choose specifically which nodes they want to use as part of their experiment. Many users select nodes in an ad-hoc manner, but there has been some effort to allow users to make more informed decisions. Services like SWORD [54] gather real-time data about resources available on nodes (e.g., CPU and memory utilization) and the network paths between pairs of nodes (e.g., path capacity and latency). Given that information, users specify node and path constraints for their desired experiment, and standard constraint satisfaction techniques are used to find a mapping [17]. Of course, there are no guarantees that resources remain at the current utilization levels after they are chosen for an experiment.

Similar ideas have been extended for use in network virtualization research. In this case, users specify full virtual network topologies that are embedded into a well-known and provisioned substrate network. Although the mapping problem is generally the same as in emulation testbeds, the goals are somewhat different. It is usually assumed that the virtual network can be embedded in multiple ways, and schedulers attempt to find the "best" solution. One approach is to use constraint satisfaction, as above, to try to find a minimum (monetary) cost embedding [42]. Other approaches include standard optimization techniques that focus on balancing load across nodes and along

network paths [80] and some work on redesigning the substrate to make the embedding problem simpler [79]. These ideas also carry over to other areas such as embedding routings in wireless sensor networks [46].

4.2 Scheduling Problem Statement

At the core of building schedules of testbed resources is finding mappings from user-defined virtual networks to a fixed physical testbed network. Not surprisingly, it is easiest to reason about network mappings using network graph structures and algorithms. Indeed, many existing graph problems are related to the testbed mapping problem, including multi-commodity flow and subgraph isomorphism. Note that the general testbed mapping problem is known to be NP-hard by reduction to the multiway separator problem [3].

User virtual networks and the physical testbed network are represented as undirected graphs. The physical testbed topology will be referred to as the *testbed* graph and be denoted $T = (V^T, E^T)$. A virtual user topology will be referred to as a *user* graph and be denoted $U = (V^U, E^U)$. Edges have capacities equal to network link capacities, denoted by $cap(e)$ for all e in E^T or E^U. Each vertex has an associated type that corresponds to a particular kind of resource in the testbed (e.g., a router or PC), denoted by $type(v)$ for all v in V^T or V^U.

The testbed graph has one special vertex type used to represent the switches and routers that indirectly connect all of the other nodes in the testbed. These *infrastructure* vertices are hidden from users and are never directly part of any user graph. Instead, they are used to form paths through T that correspond to edges in U. An example is shown in Figure 2.6, where the dashed line in each graph shows one such mapping from an edge in U to a path in T. A complete mapping from U to T is represented by $M = (M^V, M^E)$. $M^V = \{(v_1^U, v_1^T), (v_2^U, v_2^T), \ldots\}$ is a set of vertex mappings, where $v_i^U \in V^U$ and $v_i^T \in V^T$. $M^E = \{(e_1^U, \rho_1^T), (e_2^U, \rho_2^T), \ldots\}$ is the set of edge to path mappings described above, where $e_i^U \in E^U$ and $\rho_i^T = (e_1^T, e_2^T, \ldots, e_k^T), e_i^T \in E^T$, is a path in T. Every mapping must be *consistent* with the associated user graph and testbed graph. This means that every edge and vertex in U is contained in the mapping, and that the endpoints of each edge in U must be mapped to the endpoints

of the corresponding path in T:

$$\forall (v^U, w^U) \in E^U,\ (v^U, v^T), (w^U, w^T) \in M^V \iff \tag{4.1}$$

$$((v^U, w^U), \rho^T) \in M^E \text{where } \rho^T = ((v^T, z_1), \ldots, (z_m, w^T))$$

Recall that users request a reservation for resources in advance. This request is defined as $R = (U, A, l)$, where U is the user graph representing a virtual network, $A = (t_1, t_2)$ is the range of acceptable start times, and l is the length of time they need to run their experiments. If a request is accepted, it can not be moved in time or revoked. It is, however, permitted to remap any future reservations (i.e., those with begin times after the current time) on to a different set of physical resources. As such, the scheduler keeps an ongoing schedule of accepted requests. This schedule is represented by the set $S = \{S_1, S_2, \ldots, S_n\}$, where $S_i = (U_i, M_i, b_i, f_i)$ is one accepted request with user graph U_i, mapping M_i from U_i to T, request begin time b_i, and request finish time f_i. Naturally, the begin time must fall within the user's provided start time range and the reservation must be the correct length:

$$t_1 \leq b_i \leq t_2 \tag{4.2}$$

$$b_i < f_i = b_i + l \tag{4.3}$$

The problem, then, is to find a mapping from a new user graph to the testbed graph that is *feasible* given the current schedule. The schedule needs two properties to remain feasible. First, at any time, each vertex in T is mapped to at most one vertex over all U_i in the schedule:

$$\begin{aligned} \forall v^T \in V^T, (1)\ & (v^U, v^T) \notin M_i^V, \forall i, 1 \leq i \leq n, v^U \in V_i^U \text{or} \\ (2)\ & (v^U, v^T) \in M_i^V, \exists i, 1 \leq i \leq n, v^U \in V_i^U \text{and} \\ & (w^U, v^T) \notin M_j^V, \forall i \neq j, w^U \neq v^U \in V_j^U \end{aligned} \tag{4.4}$$

Second, at any time, the sum of all user graph edge capacities that are mapped to each edge in the testbed graph must be less than or equal to that edge's capacity:

$$\forall e^T \in E^T, cap(e^T) \geq \sum_{e \in X_e^T} cap(e), \text{ where} \tag{4.5}$$

$$X_e^T : \{e^U |\ (e^U, \rho^T) \in M_i^E,\ \exists i, 1 \leq i \leq n, \text{and } e^T \in \rho^T\}$$

The formal problem statement follows. Given a testbed graph T with typed vertices and edge capacities, a schedule $S = \{S_1, S_2, \ldots, S_n\}$ of previously accepted reservations, and a new reservation request $R = (U, A, l)$, find a new feasible schedule $S' = S \cup \{(U, M', b, f)\}$ where M' is a consistent mapping from U to T subject to conditions (4.1), (4.2), (4.3), (4.4), and (4.5). If no such mapping exists, leave S unchanged and reject R.

The next two sections describe two solutions to this problem. Both solutions are based on Mixed Integer Programs (MIPs) that are used to solve the mapping problem while trying to maximize the likelihood of accepting future reservation requests. The first solution uses a simpler MIP base that results in faster response times but is only applicable to certain testbed graphs. The second solves the problem for any testbed graph at the cost of higher response times.

4.3 Linear Testbed Graphs

The simplest testbed graph structure is a linear series of N infrastructure nodes. Each user-allocatable node is connected to exactly one of these infrastructure nodes, i.e., nodes with multiple edges are connected to the same infrastructure node. Linear testbed graphs arise naturally when using switches that are equipped with high capacity "stacking" connections. One such example is shown in Figure 4.1. Indeed, this testbed graph was the actual testbed graph for the Open Network Laboratory for a number of years. This testbed graph will also be used for the evaluation in the next section. The edge capacities shown are in Gb/s. Note that the labels next to nodes are used to indicate how many of that type of node are connected to the same infrastructure node. For example, the circle in the upper left actually represents four

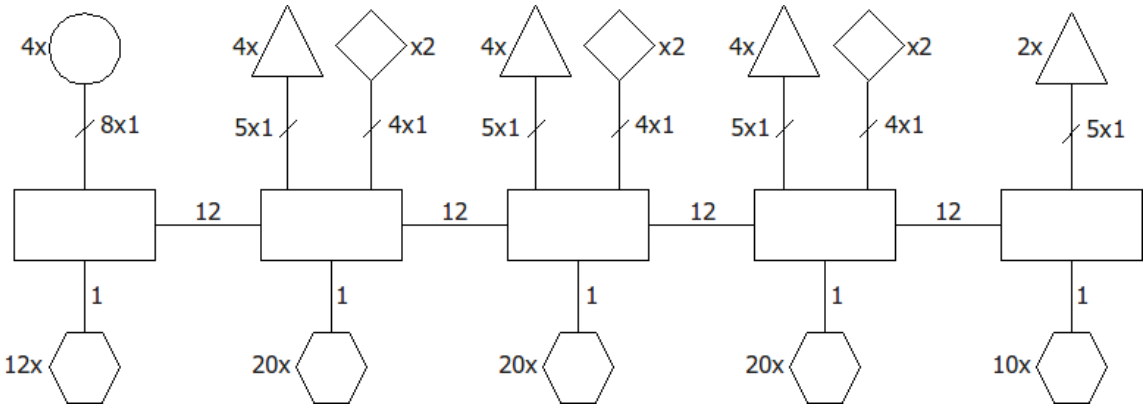

Figure 4.1: An example testbed graph.

nodes of the same type, each with eight 1 Gb/s links to the left-most infrastructure node.

Although the problem formulation specifically allows accepted reservations to be remapped to a different set of testbed nodes, the schedulers presented here do not do so. Instead, they attempt to build schedules that maximize the probability of accepting future requests. Two variations of the same basic scheduler are considered that differ in the heuristic used to achieve this goal. The first, denoted *MINBW*, minimizes usage of bandwidth between infrastructure nodes. The second, denoted *MAXPACK*, extends the first by computing *packing* scores for the user graph across all subsets of infrastructure nodes and considering potential subsets in order of the packing score. These two schedulers are clearly related, but the distinction is made in order to better characterize different approaches to the scheduling problem. Note that both of the following descriptions are applicable only for a linear testbed graph. Similar approaches would work for star and tree graphs, and more generally for any graph with only one path between any pair of nodes.

4.3.1 Minimizing Bandwidth

The pseudocode for *MINBW* is shown in Algorithm 4.1. As described in the previous section, the inputs are the user's virtual network graph U, the range of acceptable start times A, the length of the requested experiment l, the testbed graph T, and the current schedule of accepted reservations S. The first step is to compute the set

of potential begin and finish times, P, for this request. Resource availability only changes at existing reservation boundaries, so we need only consider candidate start times that correspond to the finish time of an existing reservation. For example, if no reservations overlap with the acceptable time range for U, then *computePossibleTimes* will return only one possible time in P (typically at the beginning of the range).

Algorithm 4.1 MINBW(U, A, l, T, S)

```
P ⇐ computePossibleTimes(A, l, S)
for all (b, f) ∈ P do
  O ⇐ findOverlappingReservations(b, f, S)
  T' ⇐ T − O
  if enoughNodesAvailable(U, T') then
    M ⇐ findMapping(U, T')
    if validMapping(M) then
      S ⇐ S ∪ (U, M, b, f)
      return TRUE
    end if
  end if
end for
return FALSE
```

Next, *MINBW* attempts to find a mapping from U to T for each time in P. Recall that these schedulers do not attempt to remap any previous reservations. As such, *findOverlappingReservations* is used to get the set of reservations that intersect (b, f), and then all resources from the overlapping reservations are removed from T to form T'. Here, subtraction means that all testbed nodes from reservations in O are removed from V^T and all capacity used along edges in O is subtracted from the edge capacities in E^T. *enoughNodesAvailable* is called next to check that there are at least as many nodes of each type in T' as in U. If there are not, then there is no reason to continue. In fact, even a scheduler that attempts to remap other reservations can never accept U at this point because there are not enough nodes of each type left. This will be used in the evaluation section to give bounds on the rejection rate of the scheduler.

If there are enough nodes of each type in T', then *findMapping* is called to try to find a mapping from U to T'. If a valid mapping is found, then that mapping is added to the schedule. Otherwise the next candidate time is tested, until either a valid mapping is found or there are no more times to try.

findMapping is the Mixed Integer Program that is the core of these linear graph schedulers. The basic idea is to directly incorporate the structure of the testbed graph into the MIP and then have the MIP compute a mapping from the user graph to the testbed graph directly. For both schedulers presented here, *findMapping* uses an objective function that minimizes the bandwidth used on the edges between infrastructure nodes. The *findMapping* pseudocode is shown in Algorithm 4.2.

Algorithm 4.2 findMapping(U, T)

Let N be the number of infrastructure nodes in T
Let $e_i^T, 1{\leq}i{\leq}N{-}1$, be the edge connecting infrastructure nodes i and $i+1$
Let L be the number of distinct node types in U
Let $\alpha_j, 1{\leq}j{\leq}L$, be the set of nodes of type j in U
Let $\beta_{ij}, 1{\leq}j{\leq}L, 1{\leq}i{\leq}N$, be the number of nodes of type j on infrastructure node i

Variables: $\forall u{\in}V^U$ and $\forall i, 1{\leq}i{\leq}N, \quad I_i(u){\in}\{0,1\}$
$\forall e{\in}E^U$ and $\forall i, 1{\leq}i{\leq}N{-}1, \quad x_i(e){\geq}0$

Objective: $\min \sum_{e \in E^U} \sum_{i=1}^{N-1} x_i(e)$

s.t. $\forall i, 1{\leq}i{\leq}N{-}1, \sum_{e \in E^U} x_i(e) \leq cap(e_i^T)$

$\forall u{\in}V^U, \sum_{i=1}^{N} I_i(u) = 1$

$\forall j, 1{\leq}j{\leq}L,$ and $\forall i, 1{\leq}i{\leq}N, \sum_{u \in \alpha_j} I_i(u) \leq \beta_{ij}$

$\forall e{=}(u,v){\in}E^U,$ and $\forall i, 1{\leq}i{\leq}N{-}1,$

$$x_i(e) = \left| cap(e) \left(\sum_{k=1}^{i} I_k(u) + \sum_{k=i+1}^{N} I_k(v) \right) - cap(e) \right|$$

There are a number of constants used in the MIP formulation that are derived from U and T. N is the number of infrastructure nodes in the testbed graph. $N{=}5$ for the example graph shown in Figure 4.1. Recall that only linear testbed graphs are being considered, where the N infrastructure nodes are connected in a line by $N{-}1$ edges. These edges are referred to as *infrastructure edges*. e_i^T is the edge from infrastructure node i to infrastructure node $i{+}1$. The infrastructure nodes are thus

ordered so that infrastructure nodes i and i+1 are adjacent in the linear topology. L is the number of node types in U. For example, in Figure 2.6a, there are three different node types, so L=3. The MIP will have to ensure that the number of each type mapped to individual infrastructure nodes does not exceed the number available on that infrastructure node. α_j, then, is the set of nodes of type j in U, and β_{ij} is the number of nodes of type j available on infrastructure node i.

There are only two sets of variables in the MIP. First, $I_i(u)$ is a binary variable that is used to indicate which infrastructure node each $u \in V^U$ is mapped to. That is, $I_i(u)$=1 if and only if u is mapped to infrastructure node i, and $I_i(u)$=0 otherwise. Second, $x_i(e)$ is a positive, real-valued variable representing the bandwidth used between infrastructure node i and infrastructure node i+1 due to edge $e \in E^U$. For example, the highlighted edge to path mapping in Figure 2.6 would cause $x_1(e=(A,D))=1$, but $x_1(e=(A,B))=0$. Note that even though the $x_i(e)$ variables are allowed to be real-valued, the MIP formulation will force every $x_i(e)$ to be either zero or the edge capacity, $cap(e)$. Bandwidth used between infrastructure nodes is then minimized by setting the objective function to minimize the sum of all $x_i(e)$ values.

There are four sets of constraints that govern the MIP. The first set ensures that the total bandwidth mapped to each infrastructure edge is less than the available capacity on that edge. The second set ensures that every node in U is mapped to exactly one infrastructure node. The third set uses the α_j and β_{ij} constants to ensure that the number of each type of user node mapped to each infrastructure node is less than the number available on that node, as discussed above. All of these constraints are reflecting the conditions described in the previous section. The last set of constraints, on the other hand, ties the MIP variables together.

Each constraint from this last set is used to set the value of the associated $x_i(e)$ variable based on the mapping of $e=(u,v)$'s endpoints to infrastructure nodes. For each value of i, the constraint ensures that $x_i(e)=cap(e)$ if and only if e is mapped to a path in the testbed graph that uses infrastructure edge i, and $x_i(e)$=0 otherwise. The mapping of u and v determines that path. Consider partitioning the testbed graph along infrastructure edge i such that all infrastructure nodes 1,...,i are on the "left" and infrastructure nodes i+1,...,N are on the "right." Then, $x_i(e)=cap(e)$ if u and v are mapped to different sides of this partition. Recall that $I_k(u)$=1 indicates

Table 4.1: The values of $x_i(e)$, $e = (u, v)$, given the mapping of u and v to infrastructure nodes.

$x_i(e{=}(u,v))$	$I_k(u){=}1, \exists k{\leq}i$ u is on the "left"	$I_k(u){=}1, \exists k{>}i$ u is on the "right"
$I_k(v){=}1, \exists k{<}i{+}1$ v is on the "left"	$\|0\|$	$\|-cap(e)\|$
$I_k(v){=}1, \exists k{\geq}i{+}1$ v is on the "right"	$\|cap(e)\|$	$\|0\|$

that user node u is mapped to infrastructure node k. So, $\sum_{k=1}^{i} I_k(u)$ is 1 when u is on the left of infrastructure edge i and 0 when it is on the right. The same is true for v. There are thus four cases for the values of $I_k(u)$ and $I_k(v)$ that determine the value assigned to $x_i(e)$, as shown in Table 4.1. If both endpoints are mapped to the same side of the partition, then the total of both sums in the constraint is 1, leading to $x_i(e){=}0$. Otherwise, the end points are mapped to different sides of the partition, leading to $x_i(e){=}cap(e)$. Note that the absolute value is necessary when u is on the right and v is on the left. Of course, absolute value is not a linear function and thus these constraints can not be used directly in the MIP.

Although it is not shown in the formulation for clarity, there is a standard linear and integer programming technique that is used in *findMapping* to handle absolute values. Specifically, each $x_i(e)$ variable is replaced with a pair of auxiliary variables $x_i^+(e)$ and $x_i^-(e)$. In the last set of constraints, $x_i(e)$ is replaced with $x_i^+(e) - x_i^-(e)$ and the absolute value is no longer used on the right hand side of the equation. Both auxiliary variables are non-negative. If the right hand side evaluates to $cap(e)$, then $x_i^+(e) = cap(e)$ and $x_i^-(e) = 0$. On the other hand, if the right hand side is $-cap(e)$, then $x_i^+(e) = 0$ and $x_i^-(e) = cap(e)$. In every other constraint and the objective function, $x_i(e)$ is replaced with $x_i^+(e) + x_i^-(e)$. The objective function thus ensures that both auxiliary variables are minimized, meaning that at most one of them is non-zero. If either is non-zero, then it must be $cap(e)$. The result is that $x_i^+(e) + x_i^-(e)$ correctly holds the values specified in Table 4.1.

If *findMapping* finds a solution to the MIP, then the values of $I_i(u)$ contain a valid mapping from U to T, which is then returned to *MINBW*. By minimizing the bandwidth across infrastructure nodes, *findMapping* is attempting to conserve as much of a scarce resource as possible for future requests. If, however, the infrastructure edge capacities are high, then it is possible that there could be more contention over node usage.

4.3.2 Maximizing Packing

The second scheduler, *MAXPACK*, is derived from *MINBW* and shares the same basic structure. *MAXPACK* uses a modified heuristic based on node packing to try to increase the probability that future requests can be accepted. As in *MINBW*, *MAXPACK* does not attempt to remap any existing reservations.

The pseudocode for *MAXPACK* is given in Algorithm 4.3. Everything is the same as in *MINBW* until after *enoughNodesAvailable* is called. If there are enough nodes of each type available in T', then *findValidSubsets* computes the set Q that contains all *subsets* of T' that have at least as many nodes of each type as are in U. Here, a subset of T' is any contiguous set of infrastructure nodes and all of their connected non-infrastructure nodes. If there are N infrastructure nodes, then a subset would be defined as all infrastructure nodes k_1 to k_2, where $1 \leq k_1 \leq k_2 \leq N$, and the non-infrastructure nodes connected to them. For every subset in Q, a packing score is computed and then Q is reordered using the packing scores. Each subset is passed to *findMapping* in order of best packing score first until either a valid mapping is found or there are no more subsets. The *findMapping* method is unchanged from the version used in *MINBW*.

There are many possibilities for computing the packing score. The goal is to map the user graph onto as few infrastructure nodes as possible, thereby increasing the number of resources available on each other infrastructure node. As mentioned above, this is related to minimizing bandwidth usage across infrastructure nodes, but the results can be different. Using fewer infrastructure nodes could actually increase the bandwidth usage between infrastructure nodes in the subset, but it does decrease the usage (to zero) between infrastructure nodes in the subset and those not in the subset. This

Algorithm 4.3 MAXPACK(U, A, l, T, S)

```
P ⇐ computePossibleTimes(A, l, S)
for all (b, f) ∈ P do
  O ⇐ findOverlappingReservations(b, f, S)
  T' ⇐ T − O
  if enoughNodesAvailable(U, T') then
    Q ⇐ findValidSubsets(T')
    for all T'_i ∈ Q do
      Z_i ⇐ computePackingScore(U, T'_i)
    end for
    Q ⇐ reorderSubsetsByPackingScore(Q, Z)
    for all T'_i ∈ Q do
      M ⇐ findMapping(U, T'_i)
      if validMapping(M) then
        S ⇐ S ∪ (U, M, b, f)
        return TRUE
      end if
    end for
  end if
end for
return FALSE
```

work uses a relatively simple packing score. First, if T_i' has exactly the same number of nodes of each type as U, then T_i' has the best packing score possible. In other words, if U could be packed on to T_i' such that T_i' has no unmapped nodes afterward, then that is the best possible packing. Otherwise, subsets with more unmapped nodes have a higher score than those with fewer unmapped nodes. That is, the score is the number of unmapped nodes in the subset. This simple scoring scheme treats nodes of different types equally, but it could easily be extended to give more or less weight to certain types depending on their relative availabilities, historical usage, etc.

4.3.3 Linear Graph Evaluation

Characterizing testbed scheduler performance is difficult due to the many parameters that can affect the results. This section identifies a small number of key parameters to study and provides a framework to generate a series of user requests that put a

target average load on the scheduler. Results are given for response times to new requests and for the request rejection percentages of the two schedulers.

There are three broad categories of parameters to consider: the testbed graph, user graphs, and length and start time flexibility of requests.

The testbed graph shown in Figure 4.1 is used for these evaluations. As mentioned earlier, this was the testbed graph for the Open Network Laboratory for a period of approximately two years. During that time, the *MAXPACK* scheduler was used to handle all reservations in ONL.

In these evaluations, the base testbed graph is kept constant but the capacity of the infrastructure edges is parametrized, denoted *IEC*. The capacity of these edges is often the limiting factor that leads to rejecting requests. All of the infrastructure nodes in this graph have 48x1 Gb/s ports that are connected together with vendor-specific 12 Gb/s stacking connections, so *IEC* is usually set to 12. *IEC* values of 3, 6, 9, and ∞ are also used in the evaluation.

The various types available in the testbed affect how user graphs are generated. In particular, there are two classes of nodes: backbone nodes and edge nodes. In Figure 4.1, the circles, triangles, and diamonds represent types that are used as backbone nodes, and the hexagons represent edge nodes. In reality, the circles and triangles are two types of programmable routers, the diamonds are NetFPGAs [44], and the hexagons are PCs. There are enough PCs that the number of PCs is very rarely the limiting factor when considering a new request.

User graph topologies are randomly generated using two parameters. The first parameter is the *backbone size*, *BBS*, which is the number of backbone nodes in the graph. The *BBS* backbone nodes are chosen uniformly at random from among all nodes in the testbed graph. This ensures that no user graph uses more of any type than are available in the testbed and balances the number of different types in proportion to the number available. The chosen backbone nodes are then connected randomly to form a tree. *BBS* values will vary from 2 to 8. The second parameter is the *average backbone degree*, *ABD*, which is used to determine the final shape of the user graph. Edges are added randomly between backbone nodes until twice the number of edges divided by the number of backbone nodes is at least *ABD*. *ABD* values will range

from 1 to 5, where ABD =1 guarantees that no edges will be added beyond the initial tree. Finally, all unused network interfaces in the user graph are connected to edge nodes.

The next parameter is the user's start time *flexibility*, F. The flexibility is defined as the length of the start time range, A=(t_1, t_2), divided by the reservation length, l. That is, F=$(t_2$–$t_1)/l$, where t_1, t_2, and l are given in the same units. F will range from 0 to 3, but will default to 0 unless otherwise noted. Ideally, larger values of F will lead to lower rejection probabilities.

The final parameter is the order in which requests are processed by the scheduler, denoted O. All orderings are based on the start times of the requests. Of course, the scheduler operates in an on-line fashion and thus has no control over the request order, but it is certainly true that many scheduling problems have solutions that are greatly impacted by the ordering of input events. In the testbed scheduling context, requests are naturally in rough order of increasing start time. For comparison, three orderings are explored here. $O =$ *random* places the requests in random order with respect to their start times. $O =$ *increasing* places the requests in order of strictly non-decreasing start times, and $O =$ *decreasing* places them in order of strictly non-increasing start times.

All of these parameters can ultimately affect the performance of a scheduler, so a simple framework is needed to understand the contributions of the different parameters. This work defines a new metric that captures these effects: the testbed scheduling *load*, L. Ideally, the average load reflects the average percentage of resources used in the testbed. This is complicated by having different node types, as the percentage of each type currently in use could be substantially different, and different types could be the limiting factor at different times. Rather than use a definition of load that attempts to incorporate these nuances, we use a simple probabilistic model to generate a sequence of requests that has the desired load. Given requests with a particular value of *BBS*, average lengths of l, B total backbone nodes in the testbed graph, and a desired load of L, we compute the average time between request start times, τ, as τ=$((\mathrm{BBS}/B) * l)/L$. That is, if the average reservation has length l and uses $BBS/B\%$ of the testbed resources, then a new reservation request should start every τ time units to achieve an average load of L. All of the following experiments use

this structure to generate 10000 requests with a particular value of *BBS* and L. The average reservation duration has little impact because it is only used to compute τ such that the desired load is seen across all 10000 requests. The intervals between successive start times are randomly generated from a geometric distribution with mean τ.

Experiments were conducted that varied all of the parameters discussed above. Each experiment ran one of the schedulers with all of the parameters fixed. The scheduler processed all 10000 requests and the response times where recorded for each request. The rejection percentage for the experiment was also recorded. All of the results presented are for a load that varies from $L = 0.1$ to $L = 1.0$. Each chart shows the results for different values of one parameter, while the others parameters remained fixed. Unless otherwise noted, the following values are used as the default fixed values for each parameter: $IEC = 12$, $BBS = 2$, $ABD = 3$, $F = 0$, and $O = random$. All of these results were gathered on a Linux PC with dual quad-core processors running at 2.3 GHz and 12 GB of memory. The Gurobi [31] MIP solver was used for *findMapping*.

Response Time

Figure 4.2 shows average response time results for *MAXPACK* as the backbone size is varied from 2 to 8. First, note that the average response times are all under 130 ms. This is typical over the entire evaluation. Moreover, the maximum response time over all experiments conducted for *MAXPACK* was 572 ms. This occurred for an accepted request with *BBS*=8 and a load of 1.0. In that experiment, the average acceptance response time was 120 ms with a standard deviation of 84 ms. In a normal testbed usage scenario, this means that the maximum response time even including network transit times between the user and the testbed will be under one second.

The left chart of Figure 4.2 shows the average response times for accepted requests, and the right chart shows the same for rejected requests. There are two important trends to notice. First, the response time increases as the request size increases. The acceptance times suggest that the increase is exponential. This is consistent with a MIP-based scheduler whose size (number of variables and number of constraints) increases as the user graph size increases. The second trend is a decrease in response

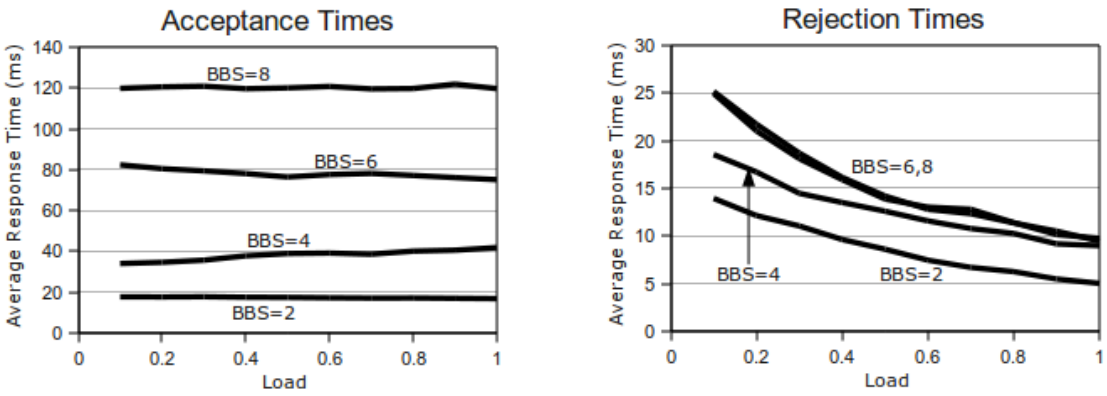

Figure 4.2: *MAXPACK* response times for different backbone sizes.

times as the load increases for rejections. This follows because there are fewer resources left in the testbed graph when the load is higher. Fewer resources lead to fewer candidate times for the request, and thus the MIP is called fewer times. This is particularly true for *MAXPACK* where there are also fewer candidate subsets to try. The results for *MINBW* are not shown, but they are similar overall. The only notable difference is that the rejection times are lower because the MIP is called fewer times for *MINBW*.

Figure 4.3 shows the average response times as the average backbone degree is varied. Note the y-axis scales (scales on all the response time charts are set differently in order to highlight the data in that chart). Clearly, *ABD* has very little effect on response times. There is a slight decrease in response times as *ABD* increases due to the nature of the user graphs. The backbone size does not change for these experiments. Higher average backbone degrees result in user graphs with fewer edges and fewer nodes. This leads to less MIP variables and constraints and thus to faster response times.

Next, Figure 4.4 shows the response times as the user's start time flexibility is varied. As expected, the response times increase as the flexibility increases. Higher flexibilities provide the scheduler more opportunities to fit the request into the schedule at the cost of running *findMapping* more times per request. The rejection response times are substantially higher for higher flexibilities. This follows because *findMapping* is potentially called many times before it is ultimately determined that the request must be rejected. The effect is less significant for higher loads because it is more likely that

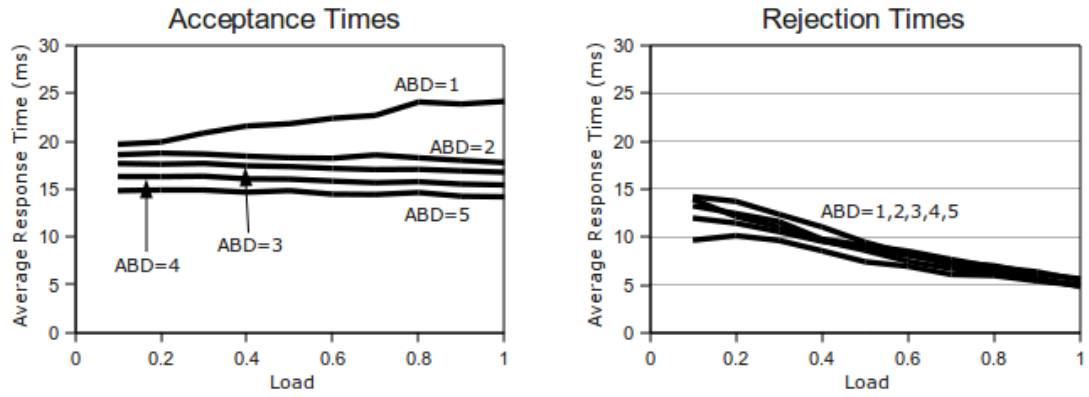

Figure 4.3: *MAXPACK* response times for different average backbone degrees.

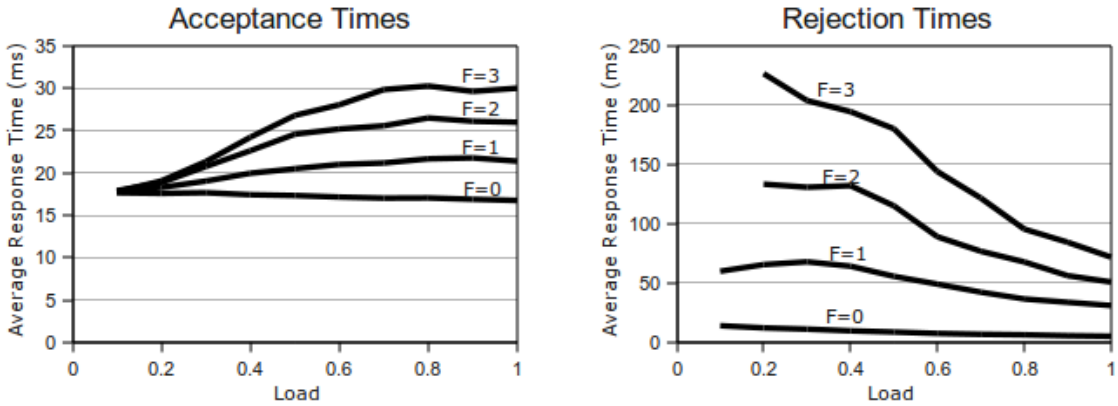

Figure 4.4: *MAXPACK* response times for different user start time flexibilities.

there are simply not enough nodes available at higher loads resulting in fewer calls to *findMapping*.

Different request orderings are evaluated next, and the results are shown in Figure 4.5. In this case, there are no substantial differences to the average response times.

Finally, the response times for different infrastructure edge capacities are shown in Figure 4.6. The acceptance times are unaffected by *IEC*. There is a slight decrease in rejection times as *IEC* increases from 3 Gb/s to 12 Gb/s. For small values of *IEC*, fewer requests are accepted because of capacity limitations, which leads to fewer nodes in use at any given time. Therefore, those requests that are rejected are less often rejected at the *enoughNodesAvailable* check for smaller values of *IEC*. This results in slightly higher average rejection response times. Of course, no requests are rejected

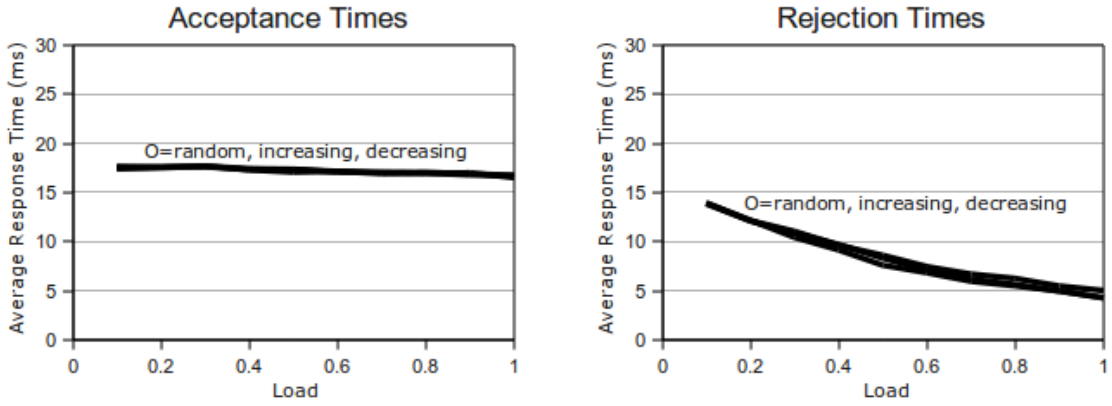

Figure 4.5: *MAXPACK* response times for different request orderings.

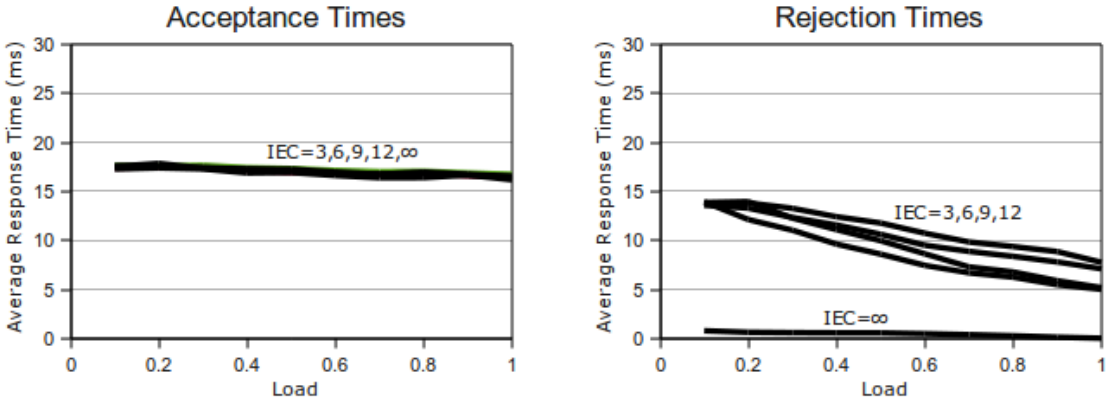

Figure 4.6: *MAXPACK* response times for different infrastructure edge capacities.

by *findMapping* when the infrastructure edge capacities are set to ∞. All rejections thus happen before *findMapping* is ever called and so the rejection times are near zero.

Rejection Rate

The second set of results concerns the rejection rate of requests. Two different rejection rates are used to reflect different aspects of the problem. The first is the *absolute rejection rate*, which is the percentage of requests rejected out of the 10000 requests in each experiment. The absolute rejection rate is useful in understanding the scheduler's overall performance from a user's perspective, i.e., how often their

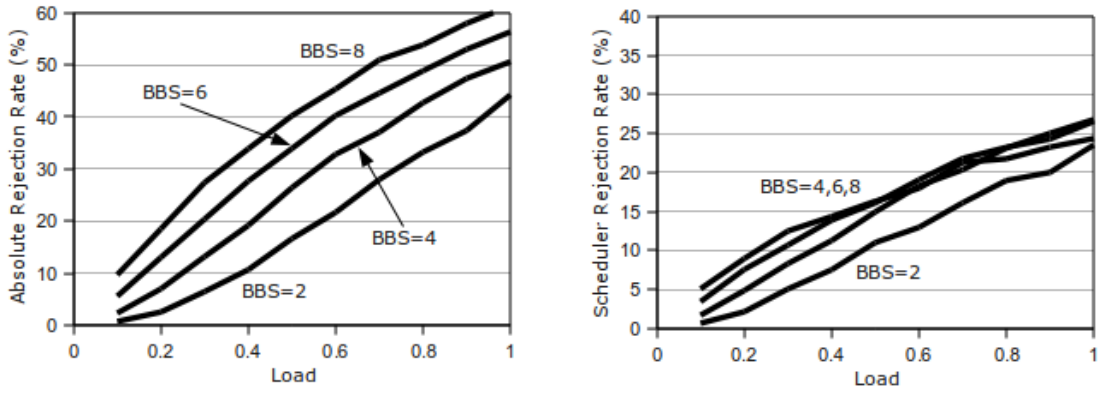

Figure 4.7: *MAXPACK* rejection rates for different backbone sizes.

requests get rejected. The second rate is the *scheduler rejection rate*. Recall that the schedulers ensure that there are enough nodes of each type available before ever trying to find a mapping. Let R_n be the number of requests that do not proceed past this check for any of the candidate times. No scheduler can ever accept such a request because there are not enough nodes of the correct types available. Let R_s be the number of requests rejected otherwise, i.e., those requests for which the scheduler attempts to find a mapping at least once. The scheduler rejection rate, then, is defined as $R_s/(10000 - R_n)$. Better schedulers will have smaller scheduler rejection rates, but even optimal schedulers may still have non-zero scheduler rejection rates if there is not enough capacity on the infrastructure edges. All results shown here are for *MAXPACK*, and the rates are similar for *MINBW*.

Figure 4.7 shows both rejection rates for requests with different backbone sizes. Clearly, increasing the size of the user graphs impacts the absolute rejection rate. As the number of backbone nodes increases, it becomes more likely that overlapping requests will overuse nodes or infrastructure edge capacity. The scheduler rejection rates do not increase substantially for larger values of *BBS*, which implies that the increases in absolute rejection rate are due to a lack of nodes rather than a lack of infrastructure edge capacity.

Next, Figure 4.8 shows the rejection rates as the average backbone degree is varied. Different values of *ABD* have very little effect on the rejection rates. The only noteworthy difference is that the scheduler rejection rate is somewhat larger when

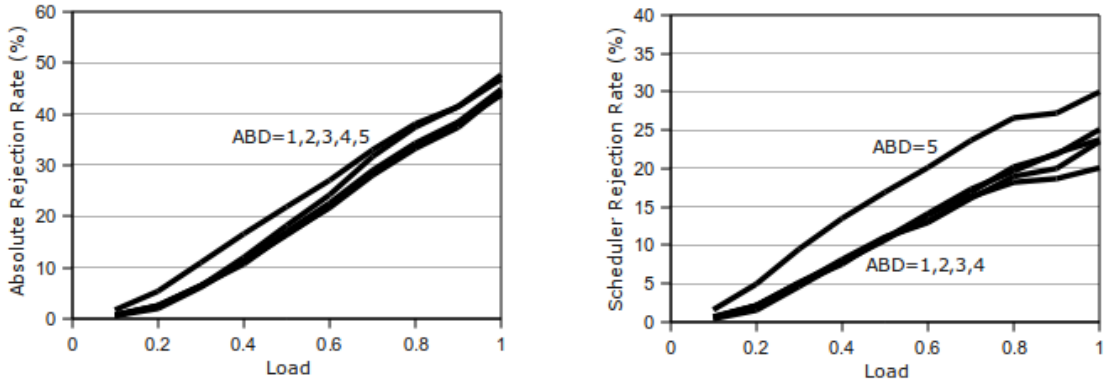

Figure 4.8: *MAXPACK* rejection rates for different average backbone degrees.

$ABD = 5$. Recall that the backbone size is fixed for these requests, so larger values of ABD result in user graphs with fewer nodes overall and more edges between backbone nodes. As such, it is less likely that the requests will be rejected at the *enoughNodesAvailable* check, which leads to higher scheduler rejection rates.

The rejection rates for different user start time flexibilities are shown in Figure 4.9. Increasing flexibility decreases the rejection rates as expected, although the decrease is minimal for $F > 1$. This is true for both the absolute and scheduler rejection rates. However, note the increase in scheduler rejection rate for $F = 2$ and $F = 3$ for high loads. The absolute rejection rate is always lower for higher flexibilities, but the scheduler rejection is actually higher in that region. To understand this, recall that the scheduler rejection rate is the percentage of rejected requests for which at least one attempt was made to find a mapping. Higher flexibilities result in more potential times that the request might fit into the schedule and thus more opportunities for *findMapping* to be called.

Figure 4.10 shows the rejection rates for different request orderings. There is no difference in rejection rates when the requests are ordered by increasing start time or decreasing start time. There is, however, a significant difference when the requests are randomly ordered. This is due to the nature of scheduling problems and how *MAXPACK* packs user requests onto as few infrastructure nodes as possible. Specifically, processing requests in start time order ensures that each new request only contends for resources with reservations that precede it in on the scheduling time line. *MAXPACK* is able to take advantage of this to pack the user graphs onto the

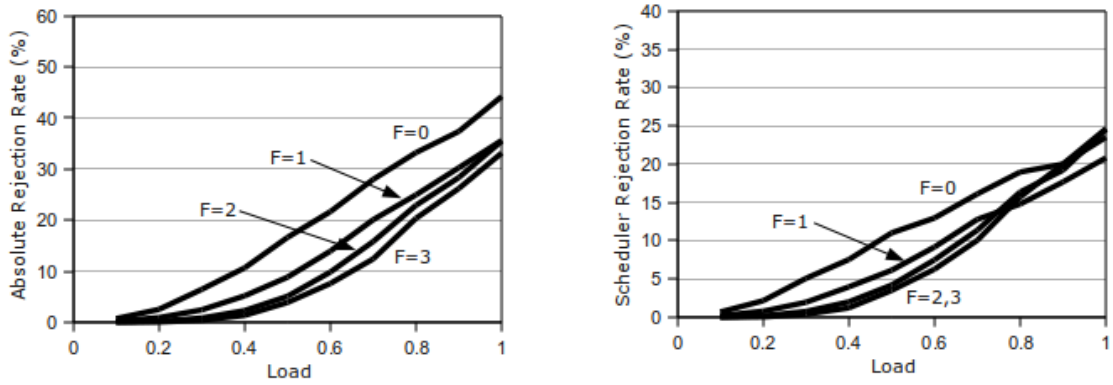

Figure 4.9: *MAXPACK* rejection rates for different user start time flexibilities.

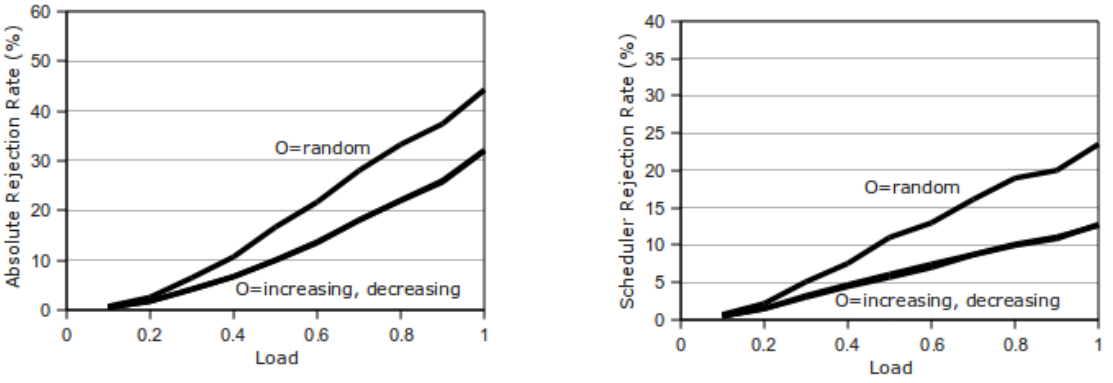

Figure 4.10: *MAXPACK* rejection rates for different request orderings.

testbed graph more efficiently. On the other hand, when requests are randomly ordered the schedule becomes filled with reservations that leave small gaps in the time line. These small gaps are not easily used later as requests that overlap them must contend with previously accepted reservations on both sides of the gap, rather than just reservations that come first in the schedule.

Finally, the infrastructure edge capacities are varied and the results are shown in Figure 4.11. Both rejection rates increase as *IEC* is decreased, as expected. $IEC = \infty$ removes any bandwidth limitations from the testbed graph, which results in a zero scheduler rejection rate, also as expected. It is interesting to note that the rates are very close for $IEC = 9$ and $IEC = 12$. This suggests that a transition in behavior occurs somewhere between values of 9 and 12. If so, it is likely that this is a transition from being fundamentally constrained by the available infrastructure edge bandwidth

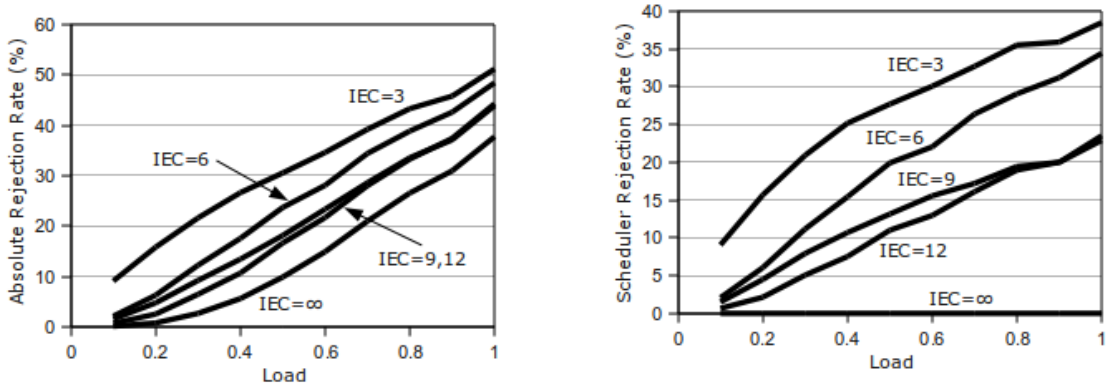

Figure 4.11: *MAXPACK* rejection rates for different infrastructure edge capacities.

to being constrained by the performance of a heuristic scheduler. The other possibility is that it is an artifact of the testbed graph that has the only 4 nodes of one type connected to the same infrastructure node. This was done because, at the time, those 4 nodes were generally only used together in practice and not in combination with the other backbone node types.

4.3.4 Discussion

This MIP-based approach works reasonably well given the parameters explored in the previous section. More importantly, the *MAXPACK* scheduler was used successfully in practice for over a year in the Open Network Laboratory. This was due in large part to the usage patterns seen in the testbed. During the summer months, the average load was small (less than 30%), and most of that load is from research projects which tend to use larger, more diverse topologies. On the other hand, the average load is higher (closer to 50%) during the fall and spring semesters when ONL is used mostly by students in networking courses. Course topologies are, however, generally much smaller. As seen in the evaluation, the scheduler can handle many smaller topologies or a few larger topologies.

Scaling up to handle higher loads is not trivial. Increasing the testbed graph size does not impact scheduler performance directly, but having more resources available allows users to build larger virtual networks. Rejection rates will be largely unaffected, assuming that user graphs increase proportionally to the testbed graph. Response

times, however, could grow to be unacceptably large because the number of variables and constraints in the MIP increases linearly with the user graph size. If there are N infrastructure nodes in the testbed graph, L distinct node types in the user graph, $|V|$ nodes in the user graph, and $|E|$ edges in the user graph, then there are $O(N|V| + (N-1)|E|)$ variables and $O((N-1) + |V| + LN + (N-1)|E|)$ constraints. Larger user graphs thus have proportionally larger MIP formulations, which leads to a potentially exponential increase in response time.

This approach could also be modified to work for a somewhat broader range of testbed graphs. Indeed, modifying the formulation for star-based testbed graphs is relatively simple. Slightly more general tree-based topologies could be supported as well. There is a fundamental limitation to this approach, however. Using indicator variables to store the mapping of user graph nodes to infrastructure nodes in the testbed allows the bandwidth used along all the associated paths to be computed directly. This assumes that there is only one path between any two nodes in the testbed graph. If there are multiple possible paths, as in a ring topology, then the current MIP formulation has no way to choose between them. This approach is therefore limited to testbed graphs that do not have multiple paths between any pair of nodes.

4.4 General Testbed Graphs

The previous section described scheduling algorithms that are applicable only to linear testbed graphs. The linear graph structure allows the scheduler to use relatively simple Mixed Integer Programs that in turn result in very fast response times. Unfortunately, many testbeds rely on more complicated testbed topologies in order to increase the number and size of concurrent user graphs. The current Open Network Laboratory testbed graph is one example that is not linear. Indeed, the ONL testbed graph is designed specifically such that there are many possible paths between every pair of nodes. Given a corresponding scheduler, these types of testbed graphs greatly increase the underlying capacity between all nodes. This section describes one such scheduler that operates on any testbed graph regardless of its structure. The only restriction is that non-infrastructure nodes with multiple edges must have all of their edges connected to the same infrastructure node.

As with the linear testbed graph schedulers, this scheduler will not attempt to remap previously accepted reservations. Rather, it takes the same approach as *MINBW* and finds mappings that minimize the capacity used between infrastructure nodes. This forces as many user nodes as possible on to common infrastructure nodes and thereby increases the likelihood that future requests can be satisfied. This generalization of *MINBW* will be denoted *GMB* (Generalized Minimum Bandwidth).

Finding mappings from user graphs to testbed graphs in the general case is closely related to other network graph problems. In particular, the mapping problem is similar to the standard multi-commodity flow problem as well as one variant known as the unsplittable flow problem. The *GMB* algorithm is based on these problems, so brief overviews of each problem will be given before describing *GMB*.

4.4.1 Related Network Flow Problems

The multi-commodity flow problem (MCF) [19] is a network flow problem where many different commodities are flowing through a shared physical infrastructure. Each commodity has one fixed point of entry into the network, one fixed point of exit from the network, and a flow capacity demand. The demand represents the "size" of the commodity in terms of the shared resources in the network. For example, the distribution of physical goods over a transportation network can be modeled as an instance of MCF. Each item (or group of similar items) is a separate commodity that is stored in one physical location and needs to be moved to another. The demand for each commodity in such problems is typically related to the shipping weight, as there are only so many trucks, train cars, or boats available to go between each pair of locations. In the networking context, many routing problems are formulated as instances of MCF, particularly when working with optical wavelength routing. The formal problem statement for MCF follows.

Given a directed graph $G = (V, E)$ where every edge $(u, v) \in E$ has capacity $cap(u, v)$ and a set of commodities K with $k_i = (s_i, t_i, d_i)$ where s_i is the source vertex, t_i is the sink vertex, and d_i is the demand for commodity i, find an assignment of flows to G for all commodities where $f_i(u, v)$ is the flow along edge (u, v) for commodity i

subject to the following constraints:

$$\forall (u,v) \in E,\ \sum_{i=1}^{|K|} f_i(u,v) \leq cap(u,v) \tag{4.6}$$

$$\forall (u,v) \in E,\ f_i(u,v) = -f_i(v,u) \tag{4.7}$$

$$\forall u \in V, u \neq s_i, t_i,\ \sum_{w \in V} f_i(u,w) = 0 \tag{4.8}$$

$$\sum_{w \in V} f_i(s_i,w) = \sum_{w \in V} f_i(w,t_i) = d_i \tag{4.9}$$

The first three constraints are slight modifications to constraints shared by all network flow problems: the capacity constraint, skew symmetry, and flow conservation. The capacity constraint, 4.6, ensures that the total flow over all commodities across each edge does not exceed the capacity of that edge. Skew symmetry, 4.7, and flow conservation, 4.8, together enforce that the per-commodity flow into a vertex equals the per-commodity flow out of that vertex, with the exception of the source and sink vertices for that commodity. Finally, 4.9 is the demand constraint that forces there to be exactly d_i units of flow in the network for commodity i.

There are many variations on this basic problem. This standard formulation is a decision problem: either all the commodities can be assigned paths in the network or not. Note that the above constraints can be used to directly formulate a Linear Program to solve MCF. In the standard formulation, there is no objective function. Some variants include edge costs that should be minimized. Others have no fixed per-commodity demands and instead attempt to maximize the amount of each commodity that can be pushed through the network.

One important characteristic of MCF is that commodity flows can be split at each vertex. The entire flow demand enters the network at s_i and leaves at t_i, but it can split arbitrarily at other vertices. For example, the flow for commodity i coming in to vertex u might be 1 unit all on one edge, but the flow out of u might be split among three edges, one with 0.5 units of flow and the other two with 0.25 units each. Another variant of considerable interest in the testbed scheduling context adds

an additional constraint that commodity flows can not be split at any vertex. This variant is known as the unsplittable flow problem (UFP) [13].

UFP is an extremely difficult problem. Even simple special cases are NP-hard. For example, the Knapsack problem can be reduced to UFP where G has a single edge. There have been some efforts to find polynomial time approximation algorithms for UFP [8, 39], but they tend to only work for special cases, such as uniform edge capacities, and have relatively poor performance bounds.

Testbed schedulers must solve a problem very similar to UFP. The virtualization techniques used to share testbed network capacity (i.e., VLANs) do not have provisions for arbitrarily splitting network flows along multiple paths. The few multipath routing techniques that exist at that level do not provide a sufficient amount of control to ensure the necessary traffic isolation among multiple users that is required in a testbed. Therefore, every user edge must be mapped to exactly one testbed network path, not split across multiple paths.

Indeed, the testbed scheduling problem is easily related to UFP by treating the testbed graph as G in UFP and mapping every edge in the user graph as a separate commodity. Unfortunately, testbed scheduling has additional constraints that can not be captured as part of UFP. Most importantly, the source and sink vertices for each user edge are not fixed and as such there is no way to assign vertices to s_i and t_i for each commodity. Finding a mapping from user nodes to testbed nodes is not orthogonal to finding a mapping between user edges and testbed paths. In other words, the testbed scheduling problem requires per-commodity source and sinks that are assigned dynamically as part of the scheduling algorithm. Once again, this further complicates an already difficult problem. The testbed scheduling problem must also handle typed nodes, which UFP (and most other network graph algorithms) do not support. The *GMB* algorithm described in the next section is still based on this idea of treating user edges as commodities in UFP along with additional constraints added to handle the differences.

4.4.2 Minimizing Bandwidth in General Graphs

The *GMB* algorithm uses the same basic structure as the *MINBW* algorithm. The pseudocode for *GMB* is shown in Algorithm 4.4. All of the steps through the call to *enoughNodesAvailable* are identical to *MINBW*. If there are enough nodes in T' to potentially satisfy the request, then *GMB* will attempt to find a mapping from U to the testbed graph. There are only two differences between *MINBW* and *GMB*. First, the original *findMapping* used in the linear algorithms has been replaced with *findGeneralMapping*. Second, there is an additional *transform* step before the mapper is called. Together, these two changes allow *GMB* to operate on any testbed graph.

Algorithm 4.4 GMB(U, A, l, T, S)

```
P ⇐ computePossibleTimes(A, l, S)
for all (b, f) ∈ P do
  O ⇐ findOverlappingReservations(b, f, S)
  T' ⇐ T − O
  if enoughNodesAvailable(U, T') then
    T'' ⇐ transform(T')
    M ⇐ findGeneralMapping(U, T'')
    if validMapping(M) then
      S ⇐ S ∪ (U, M, b, f)
      return TRUE
    end if
  end if
end for
return FALSE
```

As described in the previous section, *findGeneralMapping* is based on the unsplittable flows problem. It is a Mixed Integer Program, like the original *findMapping*, that has many of the constraints of standard MCF and UFP as well as additional constraints unique to the testbed scheduling problem. The *transform* step before *findGeneralMapping* is called is essentially a preprocessing step that prepares the testbed graph for the MIP. *transform* has two primary goals that are related. First, it modifies T' to produce a graph that is more closely related to UFP. Second, it reduces the size of T' to the minimal graph needed for *findGeneralMapping* so that the size of the MIP is as small as possible. An example of the modifications made by *transform* is shown in Figure 4.12. The edge labels are edge capacities, and different shapes represent different types of nodes. Rectangles represent infrastructure nodes.

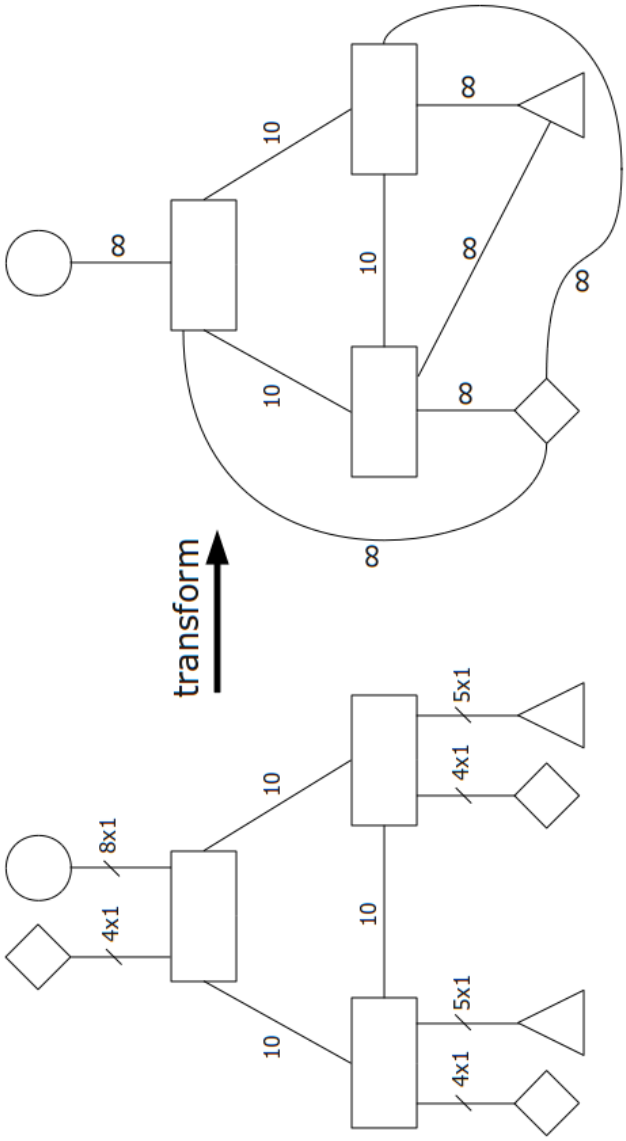

Figure 4.12: An example application of $transform$.

As Figure 4.12 shows, the *transform* function replaces every vertex of the same type with a single vertex that represents that type. These new vertices will be referred to as *type* vertices. *transform* also adds a single edge with infinite capacity from each type vertex to each infrastructure switch that had vertices of that type before the transformation. For example, the figure shows there was only one circle vertex attached to the top infrastructure node and so the circle type vertex has only one edge to that infrastructure node. On the other hand, there were diamonds connected to every infrastructure node and so the diamond type vertex has edges to every infrastructure node.

These new type vertices will serve as the source and sink vertices for all edges in the user graph. The type vertex associated with one end point of each user edge will serve as the source, and the type vertex associated with the other end point will serve as the sink. Either type vertex can serve as the source or as the sink and can be chosen differently for each user edge. Using these vertices as the sources and sinks allows every user edge to have a fixed source and sink, regardless of which vertices and edges are ultimately chosen for the mapping. The edge capacities for edges between type vertices and infrastructure nodes is set to infinity to ensure that no artificial restrictions are placed on possible mappings due to the transformation. *findGeneralMapping* will have other constraints that enforce the correct limitations based on available nodes of each type.

After the *transform* step, the *findGeneralMapping* MIP is called in order to try to find a valid mapping from U to the testbed graph. Note that one important piece of information is lost in the transformation: the number of available vertices of each type on each infrastructure node. That information is needed to ensure that any mappings found are feasible. As such, it is added as auxiliary information to T'' so that *findGeneralMapping* has access to it. The *findGeneralMapping* MIP is shown in Algorithm 4.5.

As in *findMapping*, *findGeneralMapping* relies on a small number of constants and vertex and edge sets that are derived from U and T. V_I^T is the set of all vertices in T that represent infrastructure nodes. Similarly, E_I^T is the set of all edges in T that connect one infrastructure node to another. That is, only edges that have infrastructure nodes as both end points are in E_I^T. $K = |E^U|$ is the number of edges

Algorithm 4.5 findGeneralMapping(U, T)

Let V_I^T be the set of infrastructure nodes in T
Let E_I^T be the set of edges connecting infrastructure nodes in T
Let K be the number of edges in U and (x_k, y_k) be the kth such edge with capacity cap_k
Let L be the number of distinct node types in U
Let $\alpha_j, 1{\leq}j{\leq}L$, be the set of nodes of type j in U
Let $\beta_{uj}, 1{\leq}j{\leq}L, u{\in}V_I^T$, be the number of nodes of type j on infrastructure node u
Let $\gamma_j, 1{\leq}j{\leq}L$, be the type vertex in T for type j

Variables: $\forall k, 1{\leq}k{\leq}K$ and $\forall (u,v){\in}E^T$, $f_k(u,v) \geq 0$
$\forall k, 1{\leq}k{\leq}K$ and $\forall (u,v){\in}E_I^T$, $F_k(u,v){\in}\{0,1\}$
$\forall u{\in}V_I^T$ and $\forall x{\in}V^U$, $I_u(x){\in}\{0,1\}$

Objective: $\min \sum_{k=1}^{K} \sum_{(u,v)\in E_I^T} f_k(u,v)$

s.t. $\forall (u,v){\in}E_T, \sum_{k=1}^{K}(f_k(u,v) + f_k(v,u)) \leq cap(u,v)$

$\forall k, 1{\leq}k{\leq}K,$ and $\forall u{\in}V_I^T, \sum_{w\in V^T} (f_k(u,w) - f_k(w,u)) = 0$

$\forall k, 1{\leq}k{\leq}K,$ and $\forall u{\in}V_I^T, \sum_{w\in V_I^T} F_k(u,w) \leq 1$

$\forall k, 1{\leq}k{\leq}K,$ and $\forall (u,v){\in}E_I^T, f_k(u,v) = cap_k * F_k(u,v)$

$\forall x{\in}V^U, \sum_{u\in V_I^T} I_u(x) = 1$

$\forall j, 1{\leq}j{\leq}L,$ and $\forall u{\in}V_I^T, \sum_{x\in\alpha_j} I_u(x) \leq \beta_{uj}$

$\forall k, 1{\leq}k{\leq}K,$ and $\forall j, 1{\leq}j{\leq}L,$ and $\forall u{\in}V_I^T,$

$$f_k(\gamma_j, u) = \begin{cases} cap_k * I_u(x_k) & \text{if } x_k \text{ is type } j \\ 0 & \text{otherwise} \end{cases}$$

$$f_k(u, \gamma_j) = \begin{cases} cap_k * I_u(y_k) & \text{if } y_k \text{ is type } j \\ 0 & \text{otherwise} \end{cases}$$

in U. In terms of MCF, K is also the number of commodities. Each user edge is denoted (x_k, y_k) with capacity cap_k. L, α_j, and β_{uj} are the same as in the original *findMapping*. Finally, $\gamma_j \in V^T$ is the type vertex for type j in T that was added during the *transform* step.

There are three sets of variables in the MIP. $f_k(u, v)$ comes straight from the MCF formulation and is the amount of flow for commodity k along edge $(u, v) \in E^T$. Each flow variable is positive and real-valued. The formulation forces every flow value to be either zero or the capacity of the user edge associated with that commodity. This is accomplished in part by the set of $F_k(u, v)$ variables, which are binary indicators. Each $F_k(u, v)$ variable indicates whether or not *any* flow for user edge k is assigned to testbed edge (u, v). Finally, there is second set of indicator variables, $I_u(x)$, that serves the same purpose as in *findMapping*. Namely, they indicate whether or not user node $x \in V^U$ has been assigned to infrastructure node u.

The objective function is similar to the *findMapping* objective function. The total flow used along every infrastructure edge in T is minimized. This keeps the maximum amount of capacity available for future requests and thereby increases the probability that future requests can be satisfied.

The are seven sets of constraints in the MIP, some of which come from MCF, some from UFP, and some of which are unique to testbed scheduling. The first set of constraints is the version of the MCF capacity constraint set for undirected graphs. Recall that MCF operates on directed graphs, whereas testbed scheduling uses undirected graphs. This formulation does not use any form of skew symmetry, but does assume that all flows are bidirectional. The result is that a flow along edge (u, v) implies an equal amount of flow in both directions, but $f_k(u, v) \neq f_k(v, u)$. To accounts for this, the capacity constraints explicitly ensure that the total flow for all commodities along every edge in both directions is less than the capacity of that edge.

Of course, for MCF-like behavior, there must be some notion of flow conservation across each vertex. This is represented in the second set of MIP constraints, which is the undirected version of the MCF flow conservation constraint set. These constraints force the total flow into a vertex to equal the total flow out of the vertex, on a per-commodity basis. Note that the constraints only apply to flow into and out of the

infrastructure nodes. This excludes only the type vertices, which follows because those vertices will serve as the source and sink vertices for every commodity.

The third and fourth constraint sets are derived from UFP. Specifically, the $F_k(u, v)$ indicator variables are used to ensure that all per-commodity flows are not split across multiple paths. First, constraints are added to the $F_k(u, v)$ variables to ensure that each commodity has at most one F_k indicator set for all edges going from one infrastructure node to another. That is, each commodity is only allowed to have flow along no more than one outgoing edge from each infrastructure node. Together, these F_k values form a single flow path through the infrastructure nodes for each commodity. The fourth set of constraints assigns the actual flow values according to the indicator variables. If $F_k(u, v)$ is set to one, then $f_k(u, v)$ is assigned a flow value equal to the capacity of the associated user edge. If $F_k(u, v)$ is set to zero, then so it $f_k(u, v)$.

The fifth and sixth constraint sets are the same as in the original *findMapping*. First, they ensure that every user vertex is mapped to exactly one infrastructure node in T. Second, they ensure that the number of vertices of each type on each infrastructure node is no more than the number available on that infrastructure node. Recall that β_{uj} is derived from additional data associated with T that was lost after the preprocessing *transform* step.

The final set of constraints is derived from the MCF demand constraints and is where the type vertices act as per-commodity source and sink vertices. Each user edge has two endpoints, and each endpoint is of a specific type. For example, user edge k might have endpoints of type i and j (i and j might or might not be the same). This would result in commodity k in the MIP having γ_i as its source and γ_j as its sink. Unlike standard MCF, however, this MIP must ensure that the flow for commodity k actually includes the physical nodes that were assigned as the endpoints of the associated user edge. These mappings are already contained in the $I_u(x)$ indicator variables. The first half of the last constraint set forces the flow from each commodity's source vertex through the correct infrastructure node first. Similarly, the second half forces the last edge along the flow path through the correct infrastructure to the sink vertex. In other words, there are actually only two cases. If commodity x_k is of type j and x_k is assigned to infrastructure node u, then the flow from γ_j to u is equal to

cap_k. Otherwise, the $f_k(\gamma_j, u)$ is zero, either because x_k was mapped to a different infrastructure node or because x_k is not of type j. The flow assignments for the sinks are similar.

The end result is that the two sets of indicator variables are used to map every user vertex to exactly one infrastructure node and every user edge to a single path, if there are enough resources available. The $I_u(x)$ and $f_k(u, v)$ variables contain that mapping. If it is indeed a valid mapping, then *GMB* adds it to the current schedule. Otherwise the user is notified that the request has been rejected.

Note that there are a few minor details left out of this formulation for clarity. First, many testbed graphs (such as the current ONL testbed graph) are multigraphs. That is, they have multiple edges between the same pair of infrastructure nodes. This formulation treats edges simply as pairs of vertices, which is not compatible with multigraphs. As such, the *transform* function is used to implement one standard approach to modify multigraphs in to non-multigraphs. If there are multiple edges directly connecting the same two infrastructure nodes, then *transform* inserts additional vertices in to the "middle" of each such edge. The result is that these edges are now distinguishable by their endpoints. The new vertices are treated by *findGeneralMapping* as additional infrastructure nodes.

There is one other complication related to multigraphs. The objective function minimizes flow along all infrastructure edges. If there are multiple choices for sending flow between two infrastructure nodes, then this objective function will naturally try to even the distribution of flow across all available choices. Unfortunately, this is not a desirable outcome. Consider the following example. There are only two infrastructure nodes, and there are two edges between them each with a capacity of 10. If a request arrives with two edges of capacity 1 that are each mapped to the infrastructure edges, then the objective function will assign them to different infrastructure edges resulting in a remaining capacity of 9 on each edge. If a second request then arrives with a single edge of capacity 10, then it will have to be rejected because there is not a single path with enough capacity even the total capacity is more than enough. To avoid this situation, the actual *findGeneralMapping* MIP includes edge costs for all infrastructure edges. The objective function then includes the cost of each edge as a

coefficient to the flow variables for that edge. Costs are chosen to force all possible flows along the same edge when multiple choices are available.

Finally, the testbed framework described in Chapter 2 includes support for virtual switches in user topologies in addition to virtual links. *findGeneralMapping* therefore also supports virtual switches. They are treated in a similar manner to other user nodes, i.e., a virtual switch type node is added during the *transform* step that is used as the source and sink for all edges that involve a virtual switch. They are somewhat special, however, because they are not ultimately physical nodes assigned to the user. They are assigned directly to infrastructure nodes (via the $I_u(x)$ indicator variables), but there are no restrictions on how many virtual switches can be assigned to each infrastructure node. Instead, there are additional constraints that force all vertices connected to a single virtual switch (except other virtual switches) to be mapped to the same infrastructure node as that virtual switch. This is handled by setting the indicator variables for all vertices connected to the virtual switch to match the indicator variable for the virtual switch across all infrastructure nodes.

4.4.3 General Graph Evaluation

The evaluation of *GMB* uses the same framework used for the *MINBW* and *MAXPACK* evaluations. Refer to Section 4.3.3 for the details and descriptions of all parameters. There are a small number of differences between the two evaluations. Most importantly, the testbed graph used to evaluate *GMB* is shown in Figure 4.13.

As before, the rectangles represent infrastructure nodes and the other shapes represent different node types. The two infrastructure nodes along the middle of the graph are 10 Gb/s Ethernet switches with 48 ports each. The six infrastructure nodes along the top and bottom are 1 Gb/s Ethernet switches with 48 ports each. These latter switches also have four 10 Gb/s connections each, which are used to connect to the 10 Gb/s switches.

The testbed graph is similar to the current ONL testbed graph. Indeed, the graph used for the evaluation contains everything in the actual ONL graph. Triangles represent IXPs, circles represent NSPs, diamonds represent NetFPGAs, and hexagons

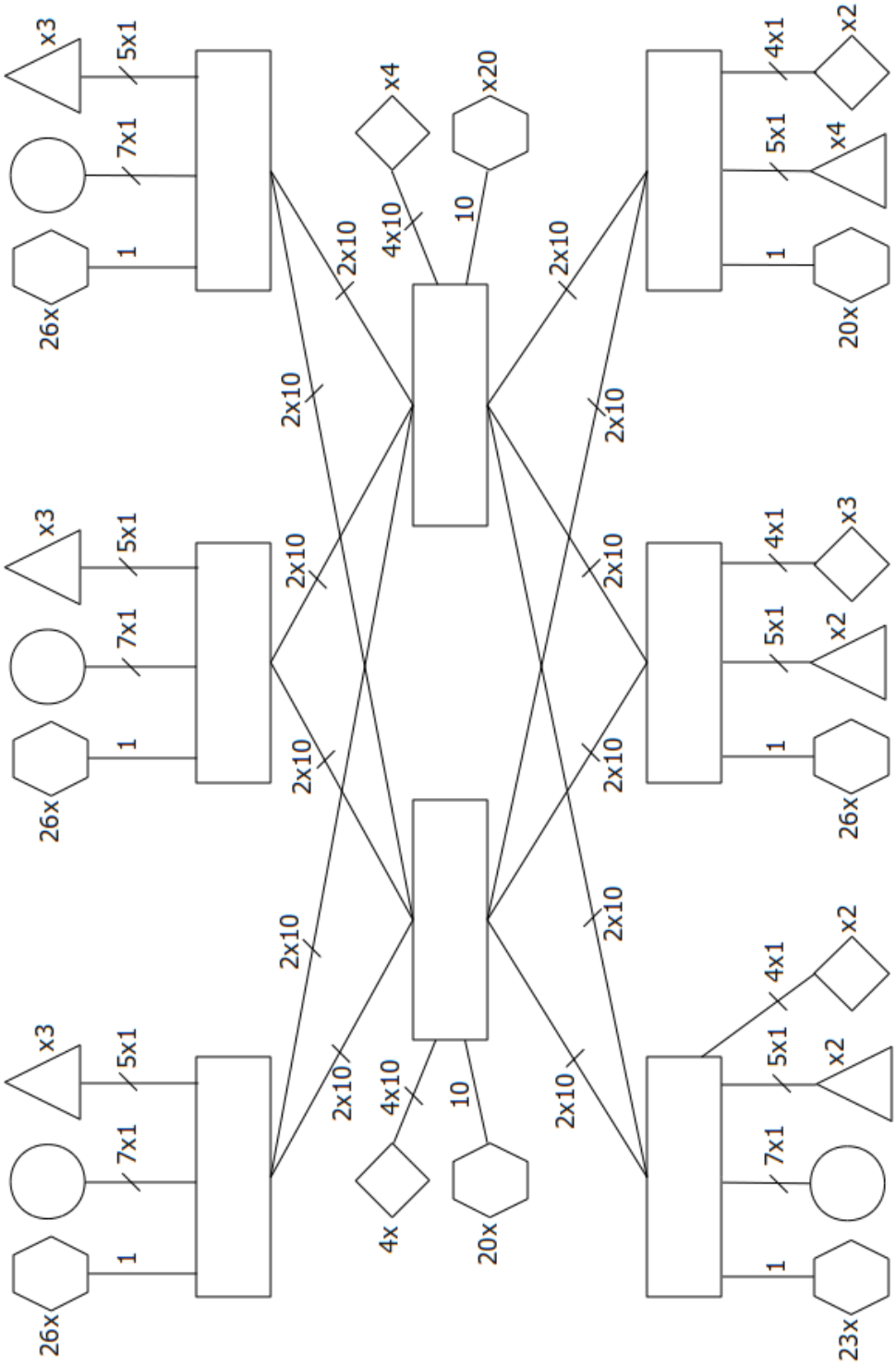

Figure 4.13: Testbed graph used in the *GMB* evaluation.

represent PCs. Note that there are actually multiple distinct types of PCs, but they have been grouped together here for simplicity. Additional nodes were added so that every port on every infrastructure switch is occupied. This included adding additional nodes of existing types as well as adding one additional type. The additional type is a representation of a newer version of the NetFPGA platform that will likely be included in ONL in the near future. Each of these nodes has four 10 Gb/s interfaces. They are also represented in the figure by diamonds. These nodes are added to the set of backbone nodes for random user graph generation. The PCs connected to the 10 Gb/s switches are used during the last step of random graph generation to connect to any used network interfaces on 10 Gb/s backbone nodes.

All of the parameters are varied as in the linear graph evaluations, with two exceptions. The values of *IEC* have changed to reflect the new testbed graph edge capacities. Specifically, *IEC* defaults to 10, and values of 5, 15, 20, and ∞ are also explored. Two request orderings are also evaluated in addition to *increasing*, *decreasing*, and *random*. $O = swapN$ is an intermediate step between the increasing start time and completely random orderings. The requests are initially ordered as in $O = increasing$. One request is then chosen uniformly at random and swapped with the request immediately after it. This random swapping is repeated a total N times, where N is the number of requests (10000 for these evaluations). This produces a request ordering that is a reasonable approximation of the orderings seen in practice. $O = swap10N$ is similar to $swapN$ except that $10N$ random swaps are performed. The other parameters remain the same, with default values of $BBS = 2$, $ABD = 3$, and $F = 0$. All of these results were gathered on a Linux PC with dual quad-core processors running at 2.3 GHz and 12 GB of memory. The Gurobi [31] MIP solver was used for *findGeneralMapping*.

Response Time

Figure 4.14 shows the average response times for *GMB* as the backbone size varies. The average responses time over all accepted requests are shown on the left, and the average response times for rejected requests are shown on the right. Note the different y-axis scales. As expected, the response times grow as the user graph size increases. For the largest user graphs, average acceptance response times are between

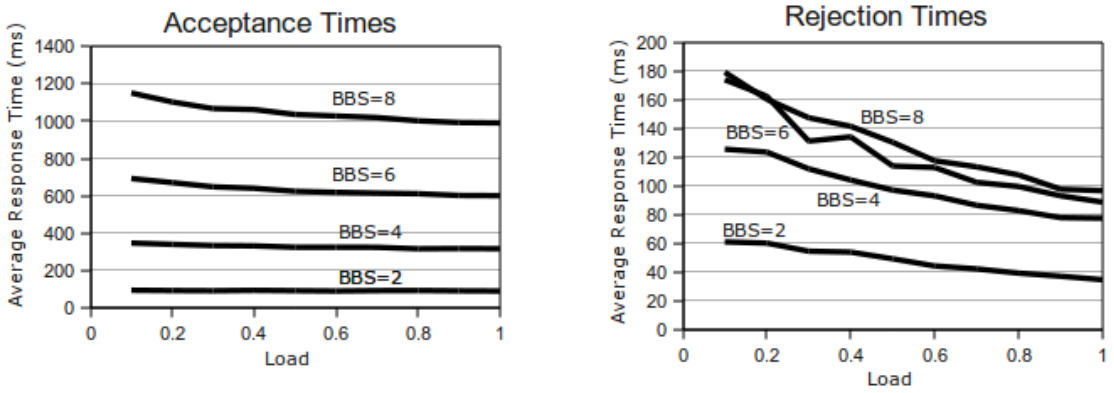

Figure 4.14: *GMB* response times for different backbone sizes.

1 and 1.2 s. The maximum response time seen over all *GMB* evaluations was 3.8 s and occurred with $BBS = 8$ and a load of 0.1. The average acceptance time in that case was 1.15 s with a standard deviation of 0.45 s and a minimum of 0.26 s. The figure suggests that the response time increase might be exponential with respect to the backbone size, which would follow naturally for any Integer Program formulation.

Overall, response times decrease as the load increases. The reasoning here is similar to that for *MAXPACK*. That is, higher loads on the testbed leave fewer unassigned resources available and therefore restrict the number of choices available to the MIP solver. This is particularly true for rejection times where the scheduler is able to determine that there are no feasible solutions more quickly. Indeed, this trend will be seen over many of the following charts as well. The other important characteristic of the rejection times shown in Figure 4.14 is that they do not continue to increase at the same rate as the user graph size increases. Once again, this follows because larger user graphs need more resources and it is less likely that all of those resources will be available at the same time.

Next, Figure 4.15 shows the responses times as the average backbone degree is varied. Note the y-axis scale differences (scales on all the response time charts are set differently in order to highlight the data in that chart). There are only minor differences in both the acceptance and rejection times for different values of *ABD*. In both cases, higher backbone degrees lead to slightly smaller responses times. This effect comes directly from the MIP formulation and user graph generation. The backbone size is fixed, and so higher values of *ABD* produce user graphs with fewer edges and fewer

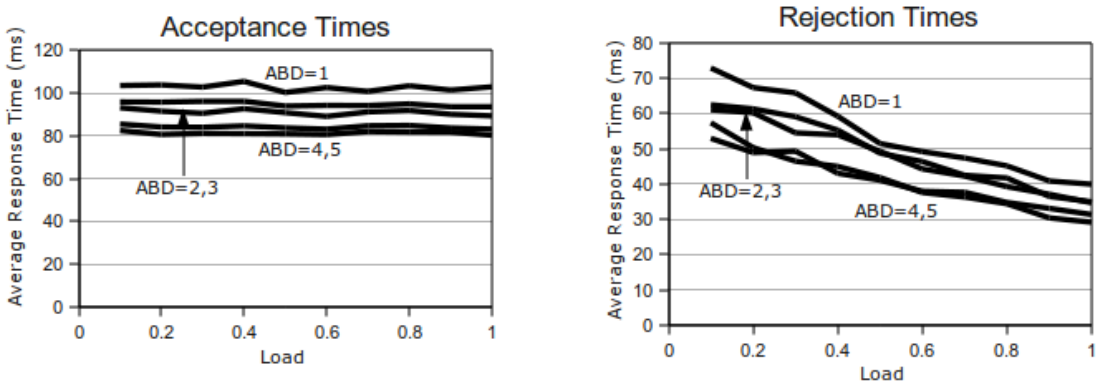

Figure 4.15: *GMB* response times for different average backbone degrees.

nodes. This results in fewer MIP variables and constraints, and therefore in faster response times.

The user's start time flexibility is varied next and the results are shown in Figure 4.16. Recall that $F=0$ implies no flexibility, i.e., that the requests specify only one possible start time. Increasing F allows the scheduler to try to fit the request into more places along the current schedule time line, at the cost of higher response times. Consider the acceptance response times for $F=1$. For low loads, the response times are very close to the case with no flexibility because there is very little resource contention and most requests are accepted immediately. The response time grows linearly up until a load of 0.8 as more and more requests must be tried with more than one start time. Finally, the response times level off for loads above 0.8, with an increase of around 40 ms over $F=0$. This time difference corresponds to the time for a single rejection, as can be seen when $F=0$ in the rejection times chart. That means that on average *findGeneralMapping* is run twice for every accepted user request. Increasing F beyond 1 does not substantially increase the acceptance times because the increased flexibility only allows a very few additional requests to be accepted.

The rejection times do increase substantially for more flexible user requests, which follows because *findGeneralMapping* will be called more times. Most striking are the results for $F=3$. For low loads, there are less overlapping reservations and so *computePossibleTimes* returns fewer potential times to try resulting in fewer calls to *findGeneralMapping*. High loads have many more potential times, but there are many less nodes available. The result is that most of the potential times are rejected

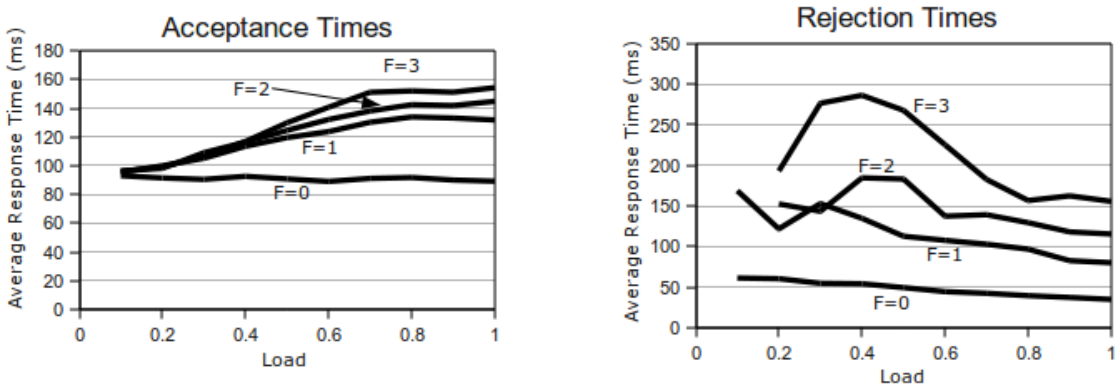

Figure 4.16: *GMB* response times for different user start time flexibilities.

at the *enoughNodesAvailable* check. Medium loads represent the worst case where *findGeneralMapping* is called most often.

Next, the response times for different request orderings are shown in Figure 4.17. The acceptance times are not affected by the order in which requests are received. The rejection times are, however, significantly affected. Rejection times for requests in increasing or decreasing start time order are near zero. As the ordering becomes more random, the rejection times increase. Recall that these are the average response times over all rejected requests. When the requests are received in order of increasing or decreasing start time, nearly all the rejections are due to the lack of available nodes. Therefore, *findGeneralMapping* is not called for most of the rejections, which results in a very low average response time. On the other hand, randomly ordering the requests causes the number of rejections to increase. In fact, these additional rejections are largely requests that were accepted when ordered by start time. For example, consider a load of 0.5. Out of the 10000 requests, ~500 were rejected because there were not enough nodes available and so *findGeneralMapping* was never called. This number does not change for different orderings. The other ~9500 requests were accepted when $O =$ *increasing*. For $O =$ *swapN*, ~9250 requests were accepted and ~250 where rejected because *findGeneralMapping* was unable to find a valid mapping given the previously accepted requests. Approximately 8350 were accepted and ~1150 were rejected by *findGeneralMapping* when $O =$ *swap10N*, and ~7600 were accepted and ~1900 were rejected by *findGeneralMapping* where $O =$ *random*. So, different request orderings only affect the average rejection response times indirectly by affecting the rejection rates. This will be examined further in the rejection rate evaluation.

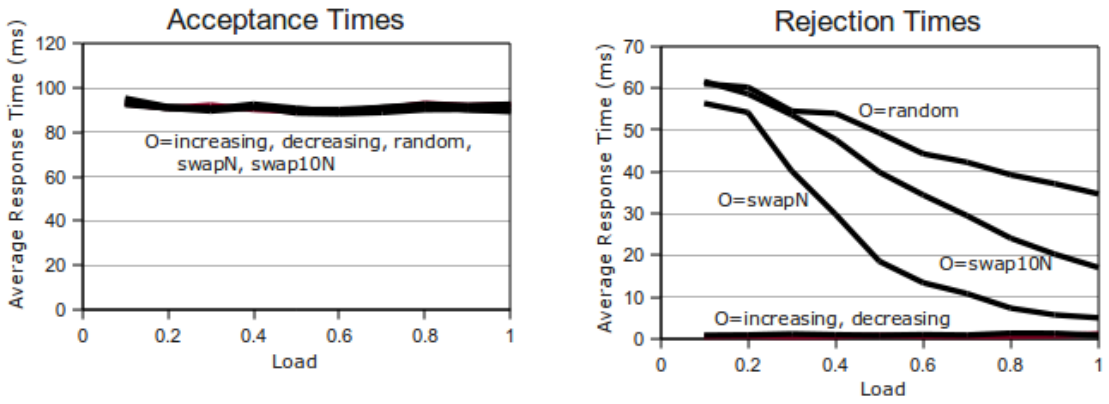

Figure 4.17: *GMB* response times for different request orderings.

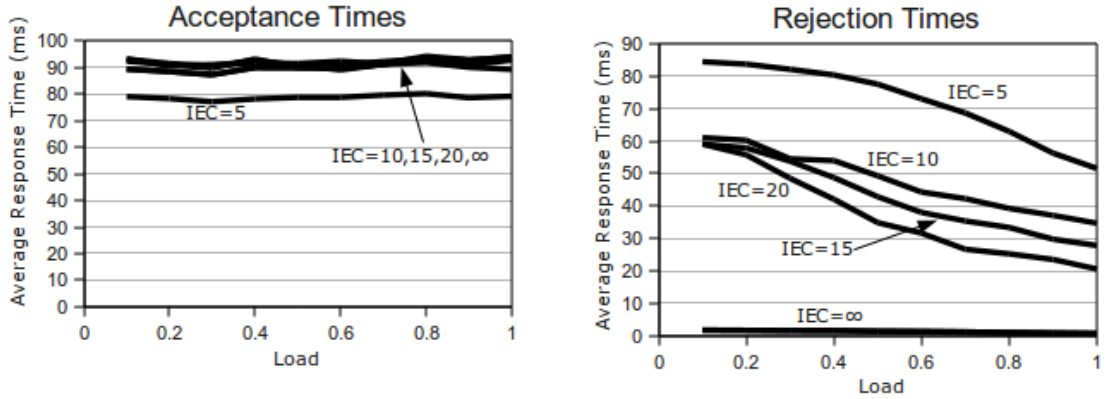

Figure 4.18: *GMB* response times for different infrastructure edge capacities.

Finally, the response times for different values of the infrastructure edge capacity are shown in Figure 4.18. The acceptance times remain largely unaffected as *IEC* changes. There is a small decrease in acceptance times when the infrastructure edge capacities are under 10 Gb/s. This is due to the nature of the requests that can be accepted in those cases. Specifically, no requests with any 10 Gb/s links can be accepted when *IEC* is less than 10 and so all accepted requests only have 1 Gb/s links. It is easier for the MIP solver to find mappings when the user edge capacities are all uniform, and therefore the average acceptance times are lower.

The rejection times fall into three groups. First, the rejection times are near zero when the infrastructure edge capacities are infinite. In that case, the only reason to reject a request is because there are not enough nodes of each type available, which is detected

immediately by *enoughNodesAvailable*. The second group is for infrastructure edge capacities of 10, 15, and 20 Gb/s. This grouping indicates that increasing the edge capacities only slightly has no impact on response times. Lastly, the rejection times are higher when the infrastructure edge capacities are less than 10 Gb/s. Again, this is because many of the rejected requests have user edge capacities that are greater than any single infrastructure edge capacity. This may seem somewhat counter-intuitive. After all, it should be easy for the scheduler to recognize that there is no way that 10 Gb/s edges can be mapped to a testbed graph that has no 10 Gb/s edges. The truth is that the MIP solver struggles with this because it always begins by finding a solution to the relaxed problem where all integer variables are treated as linear. It then uses that initial solution as a starting point to find an integer solution. In the *findGeneralMapping* formulation, this results in a relaxed solution that allows arbitrary flow splitting. From there, it takes many solver iterations to show that there is indeed no feasible solution.

Rejection Rate

The other important metric for characterizing scheduler performance is the rejection rate. This evaluation uses the same two rejection rates that were used in the *MAXPACK* and *MINBW* evaluations. Namely, the absolute rejection rate and the scheduler rejection rate. The former is the percentage of the 10000 requests that are rejected for any reason. The latter is the percentage of rejected requests for which *findGeneralMapping* was run at least once.

Figure 4.19 shows the rejection rates for *GMB* as the average backbone size is varied. The chart on the left shows the absolute rejection rates, and the chart on the right shows the scheduler rejection rates. Note the different y-axis scales. The absolute rejection rate increases as the load increases, as expected. It also increases as the backbone size increases, although the rate of increase drops as *BBS* increases. This is due to the way the user graphs are generated. As *BBS* increases for a fixed load, the number of overlapping requests decreases. The result, as intended, is that the total number of backbone nodes in use at any given time is the same regardless of the backbone size of individual requests. The increases in absolute rejection rate are

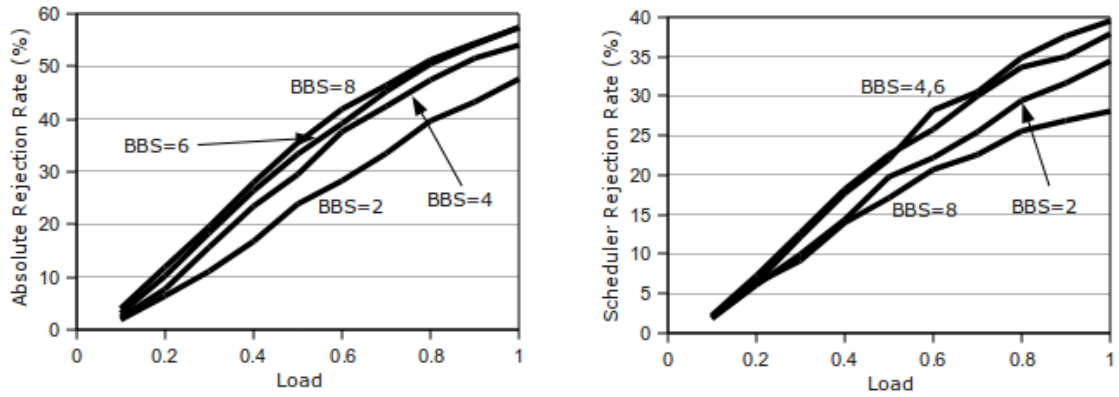

Figure 4.19: *GMB* rejection rates for different backbone sizes.

instead due to either a lack of non-backbone nodes or a lack of infrastructure edge capacity.

Indeed, these differences can be discerned from the scheduler rejection rates. As Figure 4.19 shows, the scheduler rejection rate increases from $BBS = 2$ to $BBS = 4$ then levels off for $BBS = 6$. It then drops sharply for $BBS = 8$ to rejection rates less than those for smaller requests. This is due to a shift in the primary reason that the requests were rejected. Namely, most requests with backbone sizes less than 8 are rejected because there is not enough infrastructure edge capacity, which results in a higher scheduler rejection rate. On the other hand, there are generally not enough non-backbone nodes to support larger requests and so most requests are rejected due to a lack of nodes when the backbone size is 8.

Next, Figure 4.20 shows the rejection rates for different values of the average backbone degree. Clearly, increasing or decreasing the average number of inter-backbone edges has very little impact on the rejection rate.

Figure 4.21 shows the rejection rates as the user's start time flexibility is varied. As expected, increasing the flexibility decreases the rejection rates. There are diminishing returns after the flexibility increases past 1. The key point is that by having a flexibility of only 1, users substantially increase their acceptance probability. This is particularly true for medium loads, where the absolute rejection rate is around one third of the rate when the user has no flexibility. The same general features are present for the scheduler rejection rates. Note that the same increase in scheduler

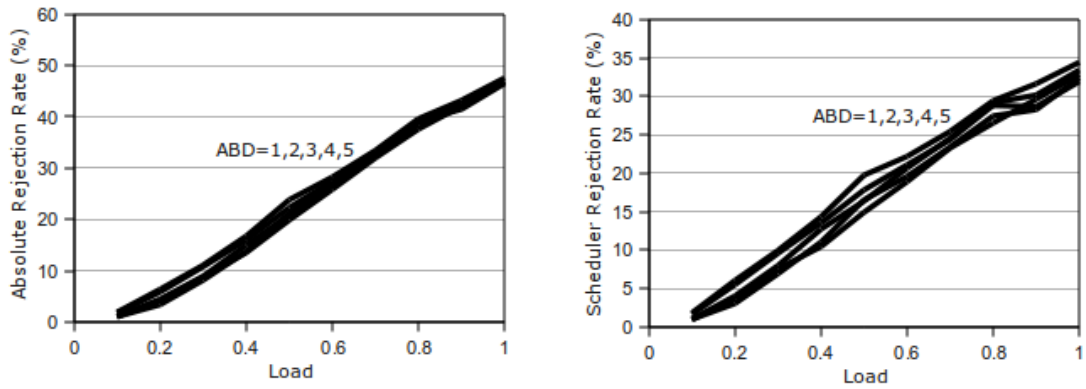

Figure 4.20: *GMB* rejection rates for different average backbone degrees.

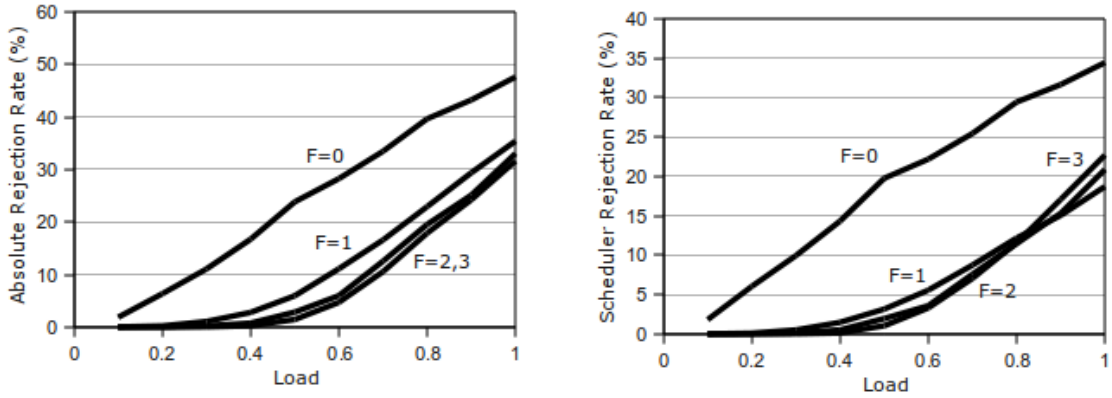

Figure 4.21: *GMB* rejection rates for different user start time flexibilities.

rejection rates for high flexibilities and high loads is present here as it was in the *MAXPACK* evaluation.

Next, the rejection rates for different request orderings are shown in Figure 4.22. As the order changes from less random to more random, the rejection rates increase. Recall from the response time evaluation that the number of requests rejected at the *enoughNodesAvailable* check is relatively constant across different orderings for a fixed load. The increased absolute rejection rates for more random orderings are thus due primarily to the inability of *findGeneralMapping* to find a mapping. This is evident in the scheduler rejection rates. Note that the same requests (same user graphs, same lengths, same start time ranges) where used in all five cases. The only difference is the order in which they are processed by *GMB*. This behavior was also

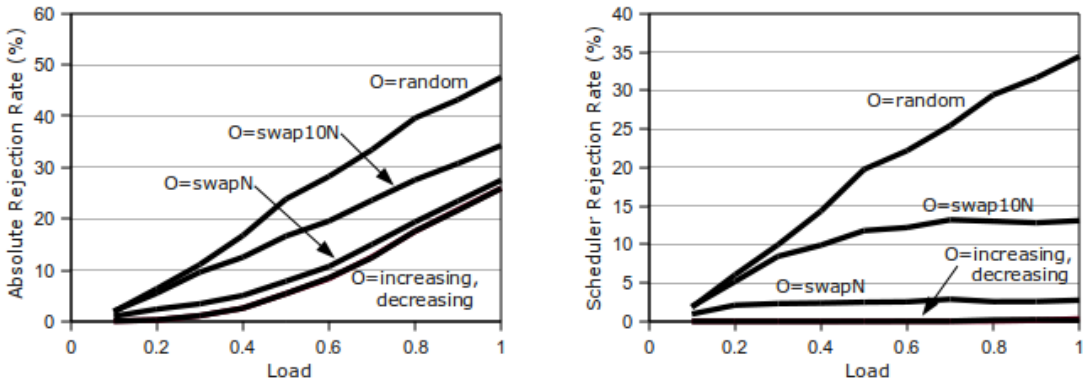

Figure 4.22: *GMB* rejection rates for different request orderings.

present in the *MAXPACK* evaluations, and the reasoning is similar here. In this case, the differences are even greater due to the possibility of user graphs with 10 Gb/s edges, as is shown next.

Figure 4.23 shows the rejection rates as the infrastructure edge capacity is varied. As expected, the rejection rates are higher for lower infrastructure edge capacities. For *IEC* less than 10 Gb/s, the rejection rates are much higher because any request with a 10 Gb/s link will be rejected by *findGeneralMapping*. Naturally, $IEC = \infty$ ensures that the scheduler rejection rate is zero. Values of *IEC* from 10 Gb/s to 20 Gb/s result in the expected proportional decreases in the scheduler rejection rates. That is, doubling *IEC* cuts the rejection rate nearly in half. Recall that the default random ordering is used here. Compared to the other parameters, increasing *IEC* has the most dramatic effect on the scheduler rejection rate under random orderings. This follows from the testbed graph used in the these evaluations. Namely, there is an abundance of infrastructure edge capacity assuming that only 1 Gb/s edges are used in user graphs. Once 10 Gb/s edges are also used, the capacity rapidly dwindles under high load. Randomly ordering the requests exacerbates this issue. Consider a request with multiple 10 Gb/s edges. When requests are ordered randomly, such a request might be processed and accepted early in the order, which results in a large percentage of the infrastructure edge capacity being used by a single request. Many subsequent overlapping requests both before and after would then be rejected. On the other hand, when $O =$ *increasing* that same request might be rejected because of the

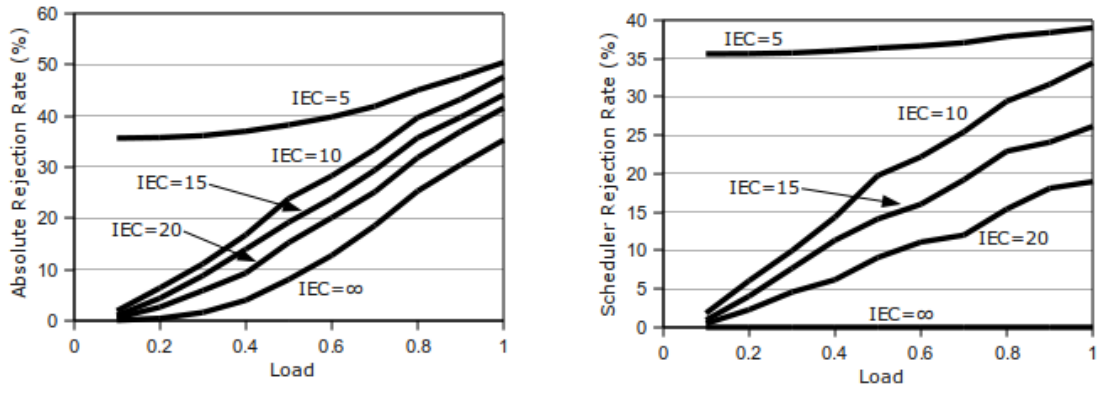

Figure 4.23: *GMB* rejection rates for different infrastructure edge capacities.

previously accepted requests. So one high capacity request has a higher probability of "blocking" many other requests under a random ordering.

Given all of these results, it is clear that user start time flexibility, request ordering, and infrastructure edge capacity all play significant roles in the rejection rates. *IEC* is not something that is easily increased in practice as it is very costly to include the additional infrastructure nodes and links necessary for such an increase. The ordering in which requests are received is also not something that can be controlled, but it is certainly true that the request ordering in practice is closest to the *swapN* ordering. $O = random$ was used as the default ordering for all of the above evaluations to understand the scheduler's performance under difficult circumstances. Similarly, the user's start time flexibility is determined entirely by the users, and these evaluations used $F = 0$ to characterize *GMB* given difficult requests.

One final set of evaluations was conducted that instead reflects the most common parameter settings found in testbeds like the Open Network Laboratory. For these final evaluations the user graph backbone size was varied as before, from $BBS = 2$ to $BBS = 8$. The average backbone degree and infrastructure edge capacity are unchanged from above, with $ABD = 3$ and $IEC = 10$. The request ordering and flexibility are modified, with $O = swapN$ and $F = 1$. Again, these values were chosen to evaluate *GMB* in the average practical case. The results are shown in Figure 4.24. The absolute rejection rate is under 40% even for large user requests and high loads.

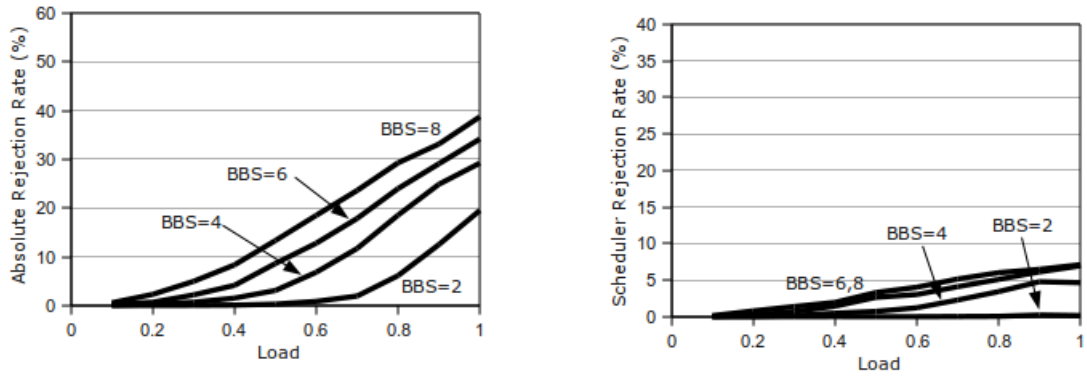

Figure 4.24: *GMB* rejection rates for common parameter settings.

Most importantly, the scheduler rejection rate is under 10% for all request sizes and loads.

4.4.4 Discussion

Overall, increasing the size of the user graphs has the most impact on response times and is clearly the factor that determines how well this approach scales. Larger testbed graphs do not directly result in substantially longer response times, but they do provide the opportunity for users to build larger requests. Of course, the limits on acceptable response times are determined primarily by how the testbed is actually used. If user sessions last only a few hours, then response times should be kept as low as possible. If user session last days or more, then it is more reasonable for the scheduler's response time to be higher. The results here show that responses are low enough for any use in medium-sized testbeds.

The rejection rates for different request orderings certainly suggest that remapping previously accepted reservations could lead to higher acceptance rates. For example, the scheduler could attempt to find a mapping for the current request as in *GMB*. If that fails, then the scheduler could temporarily remove all accepted requests that come after the current request and then attempt to map all of the outstanding requests in order. If the scheduler succeeds then the new schedule can be used as is, and if not then it could revert to the original schedule and reject the current request. Of

course, there are many possible variations on this general approach, but the main idea is to leverage the ordering characteristics as much as possible within the bounds of the testbed scheduling problem.

There are other interesting questions related to different request orderings. A scheduler with total future knowledge is naturally unaffected by ordering. Given the constraints of the testbed scheduling problem, however, even a practical optimal scheduler might perform poorly. Requests have to be accepted or rejected at request arrival time. Once the decision is made, it can not be changed. Accepted requests can also not be moved in time even if the original request had high flexibility. Schedulers do have the option of remapping nodes and links to different testbed resources for accepted requests, but even that is not absolute. The evaluation did not include a concept of the "current" time relative to the schedule, but in reality reservations that are active can not be remapped. So, it is unclear precisely how well even an optimal scheduler could perform. In the future, it would be interesting to explore this from an adversarial perspective to build worst-case orderings.

Finally, it should be noted that the rejection rates presented here are focused on whether requests were accepted or rejected. It would also be reasonable to evaluate scheduler performance in terms of overall testbed utilization rather than number of requests rejected. In other words, the scheduler could characterized by the percentage of the nodes and infrastructure capacity actually in use by the accepted requests. For example, consider a case where the scheduler could either accept two small requests or one large request. Certainly accepting two requests instead of just one is desirable, but the large request might utilize more testbed nodes than the smaller two requests together. This would lead to higher resource utility in the testbed.

Chapter 5

Conclusion

Network testbeds provide researchers, educators, and students a safe environment in which to conduct network experimentation. Researchers rely on testbeds to build and evaluate new networking systems, services, and protocols. Educators and students utilize testbeds to explore practical aspects of networking theory and to gain hands-on experience with common networking tools. In either case, testbeds can allow users to quickly configure and reconfigure experiments that span many possible network conditions.

This dissertation has explored practical and theoretical problems related to the design and implementation of network testbeds. After exploring the current state of the art for testbeds, it became clear that most existing testbeds lack resource heterogeneity. Specifically, the majority of operating testbeds are based entirely on PCs. PCs do offer a familiar platform that is capable of emulating many networking technologies, but they can not match the performance characteristics of those technologies. Moreover, they do not offer users the opportunity to interact with and learn about other networking systems. The few testbeds that do contain heterogeneous resources typically only offer limited or restricted access to those resources.

To that end, Chapter 2 presented the design of a new, more general testbed framework that naturally supports resource heterogeneity. Any type of networking device can be included in the testbed framework, from PCs to network processors to hardware routers. One of the most important features of the proposed framework is that new types of nodes can be added quickly and easily, without the need to modify any of the infrastructure software.

Once users have access to diverse types of networking devices, they also need to be able to interact with and configure those devices in an intuitive way. Of course, this can be quite challenging as different devices often have widely varied user interfaces. The new testbed framework therefore also includes a unified graphical user interface, the Remote Laboratory Interface (RLI), through which users can configure all the nodes in their experimental networks. Specifically, the RLI supports generic descriptions of the functionality available on each different type of node available in the testbed. The RLI uses these descriptions to present the user with a consistent user interface for each different type based on those descriptions.

The testbed framework also includes all of the infrastructure software needed to manage the resources in the testbed. All of the details for adding and removing user sessions, ensuring nodes are in a clean state before a new session starts, and configuring the testbed network are handled automatically. These software components treat all of the various resources generically, i.e., the software operates based on the hardware-level descriptions of each resource type. This allows new types of resources and new instances of those types to be added to the testbed with very little work. In the prototype testbed using this framework, new types and nodes can even be added without restarting any infrastructure software.

Many networking devices such as network processors, FPGAs, and PCs are capable of fulfilling many different roles in the network. For example, an FPGA might be programmed to act as a router in one experiment and as a traffic generator in another. The testbed framework also supports this notion of type extensibility at a fundamental level. Each type of resource in the testbed can have multiple specializations that correspond to different sets of functionality. This also allows the framework to provide multiple configuration interfaces targeted at users with different levels of expertise. Most importantly, new specializations can be added or existing specialization modified by any user without the need for the underlying testbed infrastructure to be modified in any way.

Given the design of this testbed framework, there is one particularly difficult and interesting problem to consider: resource sharing. Most existing testbeds employ a simple first-come first-served admission control policy. When a user wants to run an experiment, they ask for the resources for that experiment. If those resources are

available immediately, the user is granted access to the necessary nodes. If those resources are not available immediately, the user's request is rejected and they have to try again later.

The testbed framework presented in this dissertation takes a different approach. Users reserve time on the testbed in advance of when they actually wish to use the resources. For example, a user could make a reservation to use a certain set of resources for four hours starting sometime in the afternoon on the following day. Such a reservation system is a generalization of the pure admission control approach and adds a scheduling dimension to an already difficult problem. The framework uses a heuristic-based solution that relies on Mixed Integer Linear Programs to optimally solve a key subproblem based on network graph embedding. Although the solution is not optimal, it performs very well under typical testbed workloads. Moreover, the scheduler response time is often under one second, and rarely longer than a few seconds, for reasonably sized testbeds. The problem, solution, and evaluation were described in Chapter 4.

The testbed framework has also been thoroughly prototyped in the form of the Open Network Laboratory (ONL) testbed. ONL currently contains hardware routers, network processor systems, FPGA platforms, and multiple types of PCs. Most of these different types can and do support multiple specializations. Some of the types along with example specializations have been discussed in detail in Chapter 3.

ONL is open to all researchers, educators, and students. It has already been used successfully in undergraduate and graduate networking and systems courses across a small number of universities. In general, the response has been positive. Students enjoy the chance to get hands-on experience, and find that the RLI provides an intuitive interface to the testbed resources.

Although focused mostly on testbeds with fixed resources and hard performance guarantees, the framework could also be applied to other testbed environments. This includes overlay-style testbeds where each node is shared amongst many users and there are no guarantees about network capacity. Moreover, the framework could be adapted to support network configuration in other virtualized infrastructure settings. For example, application resource sharing in data centers and cloud computing environments, or Internet Service Provider management of customer Carrier Ethernet domains. In each of these cases, there are resources owned by a single entity that

are being shared by many users. The infrastructure developed for the generic testbed framework could therefore be applied to ease the ongoing configuration and management burden associated with these different virtualized systems.

In closing, there are many inherent challenges in designing and deploying testbeds that are truly useful to the networking community. The ongoing Global Environment for Network Innovations initiative is clear proof that there is a strong interest in building large-scale, heterogeneous testbeds. It has also shown that actually building such a testbed is an extremely difficult endeavor. The framework described in this dissertation has the potential to impact that work substantially. The abstractions, algorithms, and infrastructure presented here are relatively simple, yet still powerful enough to describe and configure a wide range of complex networking scenarios.

Appendix A

Source Code for the Host Specialization

This appendix contains all of the source code for one example Specialization Node Daemon. Specifically, this is the daemon counterpart to the "host" specialization of the "PC8core" base type from the Open Network Laboratory described in Section 3.1. Each of the 6 files that make up this Specialization Node Daemon are given below.

```
1  #include <iostream>
2  #include <sstream>
3  #include <cstdio>
4  #include <cstdlib>
5  #include <string>
6  #include <list>
7  #include <vector>
8  #include <map>
9  #include <exception>
10 #include <stdexcept>
11
12 #include <unistd.h>
13 #include <errno.h>
14 #include <stdlib.h>
15 #include <stdint.h>
16 #include <memory.h>
17 #include <pthread.h>
18 #include <sys/types.h>
19 #include <netinet/in.h>
20
21 #include "shared.h"
22
23 #include "host_configuration.h"
24 #include "host_globals.h"
25 #include "host_requests.h"
26
27 namespace host
28 {
29   dispatcher* the_dispatcher;
30   nccp_listener* rli_conn;
31
32   configuration* conf;
33 };
34
35 using namespace host;
36
37 int main()
38 {
39   log = new log_file("/tmp/host.log");
40   the_dispatcher = dispatcher::get_dispatcher();
41   rli_conn = NULL;
42   conf = new configuration();
```

Listing A.1: (a) host_main.cc; the main routine.

```
43    try
44    {
45      rli_conn = new nccp_listener("127.0.0.1", Default_ND_Port);
46    }
47    catch(std::exception& e)
48    {
49      write_log(e.what());
50      exit(1);
51    }
52
53    register_req<configure_node_req>(NCCP_Operation_CfgNode);
54
55    register_req<add_route_req>(HOST_AddRoute);
56    register_req<delete_route_req>(HOST_DeleteRoute);
57
58    rli_conn->receive_messages(false);
59
60    pthread_exit(NULL);
61  }
```

Listing A.1: (b) host_main.cc; the main routine.

```
1  #ifndef _HOST_GLOBALS_H
2  #define _HOST_GLOBALS_H
3
4  namespace host
5  {
6    extern dispatcher* the_dispatcher;
7    extern nccp_listener* rli_conn;
8    extern configuration* conf;
9  }; // namespace host
10
11 #endif // _HOST_GLOBALS_H
```

Listing A.2: host_globals.h; a header file containing all global variable declarations.

```
#ifndef _HOST_REQUESTS_H
#define _HOST_REQUESTS_H
namespace host
{
  class configure_node_req : public configure_node
  {
    public:
      configure_node_req(uint8_t *mbuf, uint32_t size);
      virtual ~configure_node_req();
      virtual bool handle();
  }; // class configure_node_req

  static const NCCP_OperationType HOST_AddRoute = 73;
  class add_route_req : public rli_request
  {
    public:
      add_route_req(uint8_t *mbuf, uint32_t size);
      virtual ~add_route_req();
      virtual void parse();
      virtual bool handle();

    protected:
      uint32_t prefix;
      uint32_t mask;
      uint32_t output_port;
      uint32_t nexthop_ip;
      uint32_t stats_index;
  }; // class add_route_req

  static const NCCP_OperationType HOST_DeleteRoute = 75;
  class delete_route_req : public rli_request
  {
    public:
      delete_route_req(uint8_t *mbuf, uint32_t size);
      virtual ~delete_route_req();
      virtual void parse();
      virtual bool handle();

    protected:
      uint32_t prefix;
      uint32_t mask;
  }; // class delete_route_req
};
#endif // _HOST_REQUESTS_H
```

Listing A.3: host_requests.h; the header file containing class definitions for the requests this Specialization Node Daemon supports.

```
1  #include <iostream>
2  #include <sstream>
3  #include <cstdio>
4  #include <cstdlib>
5  #include <string>
6  #include <list>
7  #include <vector>
8  #include <map>
9  #include <exception>
10 #include <stdexcept>
11
12 #include <unistd.h>
13 #include <errno.h>
14 #include <stdlib.h>
15 #include <stdint.h>
16 #include <memory.h>
17 #include <pthread.h>
18 #include <sys/types.h>
19 #include <sys/stat.h>
20 #include <netinet/in.h>
21
22 #include "shared.h"
23
24 #include "host_configuration.h"
25 #include "host_globals.h"
26 #include "host_requests.h"
27
28 using namespace host;
29
30 configure_node_req::configure_node_req(uint8_t *mbuf, uint32_t size):
31 configure_node(mbuf, size)
32 {
33 }
34
35 configure_node_req::~configure_node_req()
36 {
37 }
38
39 bool
40 configure_node_req::handle()
41 {
42   write_log("configure_node_req::handle()");
43
44   conf->set_port_info(node_conf.getPort(), node_conf.getIPAddr(),
45                                            node_conf.getNHIPAddr());
```

Listing A.4: (a) host_requests.cc; the source file containing classes for the requests this Specialization Node Daemon supports.

```
46    crd_response* resp = new crd_response(this, NCCP_Status_Fine);
47    resp->send();
48    delete resp;
49
50    return true;
51  }
52
53  add_route_req::add_route_req(uint8_t *mbuf, uint32_t size):
54  rli_request(mbuf, size)
55  {
56  }
57
58  add_route_req::~add_route_req()
59  {
60  }
61
62  bool
63  add_route_req::handle()
64  {
65    write_log("add_route_req::handle()");
66
67    std::string prefixstr = conf->addr_int2str(prefix);
68    if(prefixstr == "")
69    {
70      write_log("add_route_req: handle(): got bad prefix");
71      rli_response* rliresp = new rli_response(this, NCCP_Status_Failed);
72      rliresp->send();
73      delete rliresp;
74      return true;
75    }
76
77    std::string nhstr = conf->addr_int2str(nexthop_ip);
78    if(nhstr == "")
79    {
80      write_log("add_route_req: handle(): got bad next hop");
81      rli_response* rliresp = new rli_response(this, NCCP_Status_Failed);
82      rliresp->send();
83      delete rliresp;
84      return true;
85    }
```

Listing A.4: (b) host_requests.cc; the source file containing classes for the requests this Specialization Node Daemon supports.

```
86    NCCP_StatusType stat = NCCP_Status_Fine;
87    if(!conf->add_route(port, prefixstr, mask, nhstr))
88    {
89      stat = NCCP_Status_Failed;
90    }
91
92    rli_response* rliresp = new rli_response(this, stat);
93    rliresp->send();
94    delete rliresp;
95
96    return true;
97  }
98
99  void
100 add_route_req::parse()
101 {
102   rli_request::parse();
103
104   prefix = params[0].getInt();
105   mask = params[1].getInt();
106   output_port = params[2].getInt();
107   nexthop_ip = params[3].getInt();
108   stats_index = params[4].getInt();
109 }
110
111 delete_route_req::delete_route_req(uint8_t *mbuf, uint32_t size):
112 rli_request(mbuf, size)
113 {
114 }
115
116 delete_route_req::~delete_route_req()
117 {
118 }
```

Listing A.4: (c) host_requests.cc; the source file containing classes for the requests this Specialization Node Daemon supports.

```
119 bool
120 delete_route_req::handle()
121 {
122   write_log("delete_route_req::handle()");
123
124   std::string prefixstr = conf->addr_int2str(prefix);
125   if(prefixstr == "")
126   {
127     write_log("delete_route_req: handle(): got bad prefix");
128     rli_response* rliresp = new rli_response(this, NCCP_Status_Failed);
129     rliresp->send();
130     delete rliresp;
131     return true;
132   }
133
134   NCCP_StatusType stat = NCCP_Status_Fine;
135   if(!conf->delete_route(prefixstr, mask))
136   {
137     stat = NCCP_Status_Failed;
138   }
139
140   rli_response* rliresp = new rli_response(this, stat);
141   rliresp->send();
142   delete rliresp;
143
144   return true;
145 }
146
147 void
148 delete_route_req::parse()
149 {
150   rli_request::parse();
151
152   prefix = params[0].getInt();
153   mask = params[1].getInt();
154 }
```

Listing A.4: (d) host_requests.cc; the source file containing classes for the requests this Specialization Node Daemon supports.

```
1  #ifndef _HOST_CONFIGURATION_H
2  #define _HOST_CONFIGURATION_H
3
4  namespace host
5  {
6    typedef struct _port_info
7    {
8      std::string nic;
9      std::string ip_addr;
10     std::string next_hop;
11   } port_info;
12
13   class configuration
14   {
15     public:
16       configuration();
17       ~configuration();
18
19       void set_port_info(uint32_t portnum, std::string ip,
20                                            std::string nexthop);
21
22       bool add_route(uint16_t portnum, std::string prefix,
23                      uint32_t mask, std::string nexthop);
24       bool delete_route(std::string prefix, uint32_t mask);
25
26       std::string addr_int2str(uint32_t addr);
27
28     private:
29       static const uint32_t max_port = 0;
30       port_info port[max_port+1];
31
32       int system_cmd(std::string cmd);
33   };
34 };
35
36 #endif // _HOST_CONFIGURATION_H
```

Listing A.5: host_configuration.h; the header file containing class definitions for a singleton Linux PC configuration class.

```
1  #include <iostream>
2  #include <sstream>
3  #include <cstdio>
4  #include <cstdlib>
5  #include <string>
6  #include <list>
7  #include <vector>
8  #include <map>
9  #include <exception>
10 #include <stdexcept>
11
12 #include <unistd.h>
13 #include <errno.h>
14 #include <stdlib.h>
15 #include <stdint.h>
16 #include <memory.h>
17 #include <pthread.h>
18 #include <sys/types.h>
19 #include <sys/stat.h>
20 #include <sys/socket.h>
21 #include <netinet/in.h>
22 #include <arpa/inet.h>
23
24 #include "shared.h"
25
26 #include "host_configuration.h"
27 #include "host_globals.h"
28
29 using namespace host;
30
31 configuration::configuration()
32 {
33   port[0].nic = "data0";
34 }
35
36 configuration::~configuration()
37 {
38 }
```

Listing A.6: (a) host_configuration.cc; the source file containing classes for a singleton Linux PC configuration class.

```
39 void
40 configuration::set_port_info(uint32_t portnum, std::string ip,
41                                                 std::string nexthop)
42 {
43   std::string logstr = "configuration::set_port_info(): port="
44                      + int2str(portnum);
45   write_log(logstr);
46
47   if(portnum > max_port) return;
48
49   port[portnum].ip_addr = ip;
50   port[portnum].next_hop = nexthop;
51
52   if(port[portnum].next_hop == "0.0.0.0")
53   {
54     system_cmd("/sbin/ifconfig " + port[portnum].nic + " down");
55     system_cmd("/sbin/ifconfig " + port[portnum].nic + " "
56                + port[portnum].ip_addr + " netmask 255.255.255.0");
57     system_cmd("/sbin/ifconfig " + port[portnum].nic + " up");
58   }
59   else
60   {
61     system_cmd("/sbin/ifconfig " + port[portnum].nic + " down");
62     system_cmd("/sbin/ifconfig " + port[portnum].nic + " "
63                + port[portnum].ip_addr + " netmask 255.255.255.240");
64     system_cmd("/sbin/ifconfig " + port[portnum].nic + " up");
65   }
66
67   system_cmd("echo 14400 > /proc/sys/net/ipv4/neigh/" + port[portnum].nic
68              + "/gc_stale_time");
69   system_cmd("arping -q -c 3 -A -I " + port[portnum].nic + " "
70              + port[portnum].ip_addr);
71 }
72
73 bool
74 configuration::add_route(uint16_t portnum, std::string prefix,
75                          uint32_t mask, std::string nexthop)
76 {
77   std::string cmd = "/sbin/route add -net " + prefix + "/" + int2str(mask);
78   if(nexthop == "0.0.0.0")
79   {
80     cmd += " dev " + port[portnum].nic;
81   }
```

Listing A.6: (b) host_configuration.cc; the source file containing classes for singleton Linux PC configuration class.

```
82    else
83    {
84      cmd += " gw " + nexthop;
85    }
86    if(system_cmd(cmd) != 0) { return false; }
87    return true;
88  }
89
90  bool
91  configuration::delete_route(std::string prefix, uint32_t mask)
92  {
93    std::string cmd = "/sbin/route del -net " + prefix + "/" + int2str(mask);
94    if(system_cmd(cmd) != 0) { return false; }
95    return true;
96  }
97
98  std::string
99  configuration::addr_int2str(uint32_t addr)
100 {
101   char addr_cstr[INET_ADDRSTRLEN];
102   struct in_addr ia;
103   ia.s_addr = htonl(addr);
104   if(inet_ntop(AF_INET,&ia,addr_cstr,INET_ADDRSTRLEN) == NULL)
105   {
106     return "";
107   }
108   return std::string(addr_cstr);
109 }
110
111 int
112 configuration::system_cmd(std::string cmd)
113 {
114   write_log("configuration::system_cmd(): cmd = " + cmd);
115   int rtn = system(cmd.c_str());
116   if(rtn == -1) return rtn;
117   return WEXITSTATUS(rtn);
118 }
```

Listing A.6: (c) host_configuration.cc; the source file containing classes for singleton Linux PC configuration class.

Appendix B

Specialization Description for the NPR

This appendix contains the specialization description for the Network Processor-based Router described in detail in Section 3.4.

```
<?xml version="1.0"?>
<subtype typeName="NPR" version="1">
  <hwType typeName="IXP">
    <resource>IXP.hw</resource>
  </hwType>
  <componentType>router</componentType>
  <description>IXP based router</description>
  <assigner name="statsIndex" min="1" max="127"/>
  <fields>
    <field editable="false">
      <plabel>sampling0</plabel>
      <ptype>double</ptype>
      <default>100</default>
    </field>
    <field editable="true">
      <plabel>sampling1</plabel>
      <ptype>double</ptype>
      <default>50</default>
    </field>
    <field editable="true">
      <plabel>sampling2</plabel>
      <ptype>double</ptype>
      <default>25</default>
    </field>
    <field editable="true">
      <plabel>sampling3</plabel>
      <ptype>double</ptype>
      <default>12.5</default>
    </field>
  </fields>
  <rebootCommand opcode="0" numParams="1">
    <displayLabel>RebootParameters</displayLabel>
    <description>Reboot Parameters</description>
    <param>
      <plabel>CPaddr</plabel>
      <ptype>string</ptype>
    </param>
  </rebootCommand>
  <init opcode="0" numParams="1">
    <displayLabel>UserDaemonPath</displayLabel>
    <description>UserDaemonPath</description>
    <param editable="true">
      <plabel>path</plabel>
      <ptype>string</ptype>
      <default>/users/onl/ixp/npr</default>
    </param>
  </init>
  <commands>
    <command opcode="98" numParams="3">
      <displayLabel>SetSamplingRates</displayLabel>
      <description>set sampling rate percentages</description>
      <param editable="true">
```

Listing B.1: (a) NPR.shw; specialization description for the NPR.

```
53        <plabel>sampling1</plabel>
54        <ptype>field</ptype>
55      </param>
56      <param editable="true">
57        <plabel>sampling2</plabel>
58        <ptype>field</ptype>
59      </param>
60      <param editable="true">
61        <plabel>sampling3</plabel>
62        <ptype>field</ptype>
63      </param>
64    </command>
65  </commands>
66  <monitoring>
67    <command opcode="101" numParams="1">
68      <displayLabel>StatsPreQPkt</displayLabel>
69      <description>read pre queue packet count for stats index</description>
70      <units>packets</units>
71      <param editable="true">
72        <plabel>index</plabel>
73        <ptype>int</ptype>
74      </param>
75    </command>
76    <command opcode="102" numParams="1">
77      <displayLabel>StatsPostQPkt</displayLabel>
78      <description>read post queue packet count for stats index</description>
79      <units>packets</units>
80      <param editable="true">
81        <plabel>index</plabel>
82        <ptype>int</ptype>
83      </param>
84    </command>
85    <command opcode="103" numParams="1">
86      <displayLabel>StatsPreQByte</displayLabel>
87      <description>read pre queue byte count for stats index</description>
88      <units>bytes</units>
89      <param editable="true">
90        <plabel>index</plabel>
91        <ptype>int</ptype>
92      </param>
93    </command>
94    <command opcode="104" numParams="1">
95      <displayLabel>StatsPostQByte</displayLabel>
96      <description>read post queue byte count for stats index</description>
97      <units>bytes</units>
98      <param editable="true">
99        <plabel>index</plabel>
100       <ptype>int</ptype>
101     </param>
102   </command>
103   <command opcode="105" numParams="1">
104     <displayLabel>ReadRegisterByte</displayLabel>
```

Listing B.1: (b) NPR.shw; specialization description for the NPR.

```
      <description>read byte count for register</description>
      <units>bytes</units>
      <param editable="true">
        <plabel>index</plabel>
        <ptype>int</ptype>
      </param>
    </command>
    <command opcode="106" numParams="1">
      <displayLabel>ReadRegisterPacket</displayLabel>
      <description>read packet count for register</description>
      <units>packets</units>
      <param editable="true">
        <plabel>index</plabel>
        <ptype>int</ptype>
      </param>
    </command>
  </monitoring>
  <tables>
    <table title="PluginTable" enabled="false">
      <assigner name="microengines" min="0" max="4"/>
      <fields>
      </fields>
      <commands>
        <command opcode="92" numParams="1">
          <displayLabel>PluginDebugging</displayLabel>
          <description>plugin debugging</description>
          <param editable="true">
            <plabel>file_name</plabel>
            <ptype>string</ptype>
          </param>
        </command>
      </commands>
      <monitoring>
      </monitoring>
      <entry name="plugin" numFields="2" numColumns="2" enabled="false">
        <fields>
          <field editable="true">
            <plabel>microengine</plabel>
            <ptype>int</ptype>
            <tableAssigned>microengines</tableAssigned>
          </field>
          <field editable="true">
            <plabel>path</plabel>
            <ptype>string</ptype>
            <help>the full path name of uof file including the file name</help>
          </field>
        </fields>
        <commandLog/>
        <commands>
          <addCommand opcode="85" numParams="2">
            <displayLabel>AddPlugin</displayLabel>
            <description>add plugin</description>
```

Listing B.1: (c) NPR.shw; specialization description for the NPR.

```
157             <param editable="true">
158               <plabel>microengine</plabel>
159               <ptype>field</ptype>
160             </param>
161             <param editable="true">
162               <plabel>path</plabel>
163               <ptype>field</ptype>
164             </param>
165           </addCommand>
166           <deleteCommand opcode="86" numParams="1">
167             <displayLabel>DeletePlugin</displayLabel>
168             <description>delete plugin</description>
169             <param editable="false">
170               <plabel>microengine</plabel>
171               <ptype>field</ptype>
172             </param>
173           </deleteCommand>
174           <command opcode="89" numParams="2">
175             <displayLabel>PluginCommand</displayLabel>
176             <description>plugin command</description>
177             <param editable="false">
178               <plabel>microengine</plabel>
179               <ptype>field</ptype>
180             </param>
181             <param editable="true">
182               <plabel>message</plabel>
183               <ptype>string</ptype>
184             </param>
185           </command>
186         </commands>
187         <monitoring>
188           <command opcode="87" numParams="2">
189             <displayLabel>PluginCounter</displayLabel>
190             <description>monitor plugin counter</description>
191             <units>packets</units>
192             <param editable="false">
193               <plabel>microengine</plabel>
194               <ptype>field</ptype>
195             </param>
196             <param editable="true">
197               <plabel>counter</plabel>
198               <ptype>int</ptype>
199             </param>
200           </command>
201         </monitoring>
202         <display>
203           <column title="plugin" width="115">
204             <fieldName>path</fieldName>
205           </column>
206           <column title="microengine" width="50">
207             <fieldName>microengine</fieldName>
208           </column>
```

Listing B.1: (d) NPR.shw; specialization description for the NPR.

```
        </display>
      </entry>
    </table>
  </tables>
  <ports>
    <fields>
    </fields>
    <commands>
    </commands>
    <monitoring>
      <command opcode="68" numParams="1">
        <displayLabel>ReadQueueLength</displayLabel>
        <description>read queue length</description>
        <units>bytes</units>
        <param>
          <plabel>queue_id</plabel>
          <ptype>int</ptype>
        </param>
      </command>
      <command opcode="107" numParams="0">
        <displayLabel>RXPKT</displayLabel>
        <description>read rx packet count</description>
        <units>packets</units>
      </command>
      <command opcode="108" numParams="0">
        <displayLabel>RXBYTE</displayLabel>
        <description>read rx byte count</description>
        <units>bytes</units>
      </command>
      <command opcode="109" numParams="0">
        <displayLabel>TXPKT</displayLabel>
        <description>read tx packet count</description>
        <units>packets</units>
      </command>
      <command opcode="110" numParams="0">
        <displayLabel>TXBYTE</displayLabel>
        <description>read tx byte count</description>
        <units>bytes</units>
      </command>
    </monitoring>
    <tables>
      <routeTable title="RouteTable">
        <fields>
          <field editable="true">
            <plabel>priority</plabel>
            <ptype>int</ptype>
            <default>56</default>
            <updateCommand opcode="91">
              <displayLabel>Set Table Priority</displayLabel>
              <description>priority for routing table. range (0-60)</description>
            </updateCommand>
          </field>
```

Listing B.1: (e) NPR.shw; specialization description for the NPR.

```
261         </fields>
262         <commands>
263         </commands>
264         <monitoring>
265         </monitoring>
266         <entry name="route" numFields="4" numColumns="3" enable="false">
267           <fields>
268             <field editable="true">
269               <plabel>prefix</plabel>
270               <ptype>ipaddress</ptype>
271               <default>0.0.0.0</default>
272             </field>
273             <field editable="true">
274               <plabel>mask</plabel>
275               <ptype>int</ptype>
276               <default>32</default>
277             </field>
278             <field editable="true">
279               <plabel>nexthop</plabel>
280               <ptype>nexthop</ptype>
281               <default>0</default>
282             </field>
283             <field editable="true">
284               <plabel>statsIndex</plabel>
285               <ptype>int</ptype>
286               <hwAssigned>statsIndex</hwAssigned>
287             </field>
288           </fields>
289           <commands>
290             <addCommand opcode="73" numParams="4">
291               <displayLabel>AddRoute</displayLabel>
292               <description>add_route</description>
293               <param editable="true">
294                 <plabel>prefix</plabel>
295                 <ptype>field</ptype>
296               </param>
297               <param editable="true">
298                 <plabel>mask</plabel>
299                 <ptype>field</ptype>
300               </param>
301               <param editable="true">
302                 <plabel>nexthop</plabel>
303                 <ptype>field</ptype>
304               </param>
305               <param editable="true">
306                 <plabel>statsIndex</plabel>
307                 <ptype>field</ptype>
308               </param>
309             </addCommand>
310             <deleteCommand opcode="75" numParams="3">
311               <displayLabel>DeleteRoute</displayLabel>
312               <description>delete route</description>
```

Listing B.1: (f) NPR.shw; specialization description for the NPR.

```
313               <param editable="false">
314                 <plabel>prefix</plabel>
315                 <ptype>field</ptype>
316               </param>
317               <param editable="false">
318                 <plabel>mask</plabel>
319                 <ptype>field</ptype>
320               </param>
321               <param editable="false">
322                 <plabel>nexthop</plabel>
323                 <ptype>field</ptype>
324               </param>
325             </deleteCommand>
326             <command opcode="74" numParams="4">
327               <displayLabel>Update Nexthop</displayLabel>
328               <description>update route's next hop</description>
329               <param editable="false">
330                 <plabel>prefix</plabel>
331                 <ptype>field</ptype>
332               </param>
333               <param editable="false">
334                 <plabel>mask</plabel>
335                 <ptype>field</ptype>
336               </param>
337               <param editable="true">
338                 <plabel>nexthop</plabel>
339                 <ptype>field</ptype>
340               </param>
341               <param editable="false">
342                 <plabel>statsIndex</plabel>
343                 <ptype>field</ptype>
344               </param>
345             </command>
346             <command opcode="96" numParams="4">
347               <displayLabel>SetStatsIndex</displayLabel>
348               <description>set stats index for route</description>
349               <param editable="false">
350                 <plabel>prefix</plabel>
351                 <ptype>field</ptype>
352               </param>
353               <param editable="false">
354                 <plabel>mask</plabel>
355                 <ptype>field</ptype>
356               </param>
357               <param editable="false">
358                 <plabel>nexthop</plabel>
359                 <ptype>field</ptype>
360               </param>
361               <param editable="true">
362                 <plabel>statsIndex</plabel>
363                 <ptype>field</ptype>
364               </param>
```

Listing B.1: (g) NPR.shw; specialization description for the NPR.

```
            </command>
          </commands>
          <monitoring>
            <command opcode="101" numParams="1">
              <displayLabel>StatsPreQPkt</displayLabel>
              <description>pre queue packet count for stats index</description>
              <units>packets</units>
              <param editable="false">
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
            <command opcode="102" numParams="1">
              <displayLabel>StatsPostQPkt</displayLabel>
              <description>post queue packet count for stats index</description>
              <units>packets</units>
              <param editable="false">
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
            <command opcode="103" numParams="1">
              <displayLabel>StatsPreQByte</displayLabel>
              <description>pre queue byte count for stats index</description>
              <units>bytes</units>
              <param>
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
            <command opcode="104" numParams="1">
              <displayLabel>StatsPostQByte</displayLabel>
              <description>post queue byte count for stats index</description>
              <units>bytes</units>
              <param editable="false">
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
          </monitoring>
          <display>
            <column title="prefix/mask" width="120">
              <fieldName>prefix</fieldName>
              <symbol>/</symbol>
              <fieldName>mask</fieldName>
            </column>
            <column title="nexthop" width="50">
              <fieldName>nexthop</fieldName>
            </column>
            <column title="stats" width="50">
              <fieldName>statsIndex</fieldName>
            </column>
```

Listing B.1: (h) NPR.shw; specialization description for the NPR.

```
          </display>
        </entry>
      </routeTable>
      <table title="FilterTable">
        <fields>
        </fields>
        <commands>
        </commands>
        <monitoring>
        </monitoring>
        <entry name="filter" numFields="28" numColumns="10" enabled="true">
          <fields editable="true">
            <field>
              <plabel>aux</plabel>
              <ptype>boolean</ptype>
              <default>false</default>
              <help>auxiliary filters make one copy of matching packets</help>
            </field>
            <field editable="true" width="150">
              <plabel>destination_address</plabel>
              <ptype>ipaddress</ptype>
              <default>192.168.0.0</default>
              <help>IP destination address prefix</help>
            </field>
            <field editable="true">
              <plabel>destination_mask</plabel>
              <ptype>int</ptype>
              <default>16</default>
              <help>IP destination address subnet mask (CIDR notation)</help>
            </field>
            <field editable="true" width="150">
              <plabel>source_address</plabel>
              <ptype>ipaddress</ptype>
              <default>192.168.0.0</default>
              <help>IP source address prefix</help>
            </field>
            <field editable="true">
              <plabel>source_mask</plabel>
              <ptype>int</ptype>
              <default>16</default>
              <help>IP source address subnet mask (CIDR notation)</help>
            </field>
            <field editable="true">
              <plabel>plugin_tag</plabel>
              <ptype>int</ptype>
              <wildcard>-1</wildcard>
              <default>0</default>
              <help>5-bit field that plugins can modify to force a
                    different match on reclassification</help>
            </field>
            <field editable="true">
              <plabel>protocol</plabel>
```

Listing B.1: (i) NPR.shw; specialization description for the NPR.

```
469               <ptype>string</ptype>
470               <default>tcp</default>
471               <pChoices editable="true">
472                 <pChoice>tcp</pChoice>
473                 <pChoice>udp</pChoice>
474                 <pChoice>icmp</pChoice>
475                 <pChoice>*</pChoice>
476               </pChoices>
477               <help>select from list or type in protocol number.</help>
478             </field>
479             <field editable="true">
480               <plabel>destination_port</plabel>
481               <ptype>int</ptype>
482               <wildcard>-1</wildcard>
483               <default>*</default>
484               <help>TCP/UDP destination port number, * is wildcard</help>
485             </field>
486             <field editable="true">
487               <plabel>source_port</plabel>
488               <ptype>int</ptype>
489               <wildcard>-1</wildcard>
490               <default>*</default>
491               <help>TCP/UDP source port number, * is wildcard</help>
492             </field>
493             <field editable="true">
494               <plabel>exception nonip</plabel>
495               <ptype>int</ptype>
496               <wildcard>-1</wildcard>
497               <default>0</default>
498               <pChoices editable="false">
499                 <pChoice>0</pChoice>
500                 <pChoice>1</pChoice>
501                 <pChoice>*</pChoice>
502               </pChoices>
503               <help>Flag to match non-IPv4 packets, * is wildcard</help>
504             </field>
505             <field editable="true">
506               <plabel>exception arp</plabel>
507               <ptype>int</ptype>
508               <wildcard>-1</wildcard>
509               <default>0</default>
510               <pChoices editable="false">
511                 <pChoice>0</pChoice>
512                 <pChoice>1</pChoice>
513                 <pChoice>*</pChoice>
514               </pChoices>
515               <help>Flag to match ARP packets, * is wildcard</help>
516             </field>
517             <field editable="true">
518               <plabel>exception ipopt</plabel>
519               <ptype>int</ptype>
520               <wildcard>-1</wildcard>
```

Listing B.1: (j) NPR.shw; specialization description for the NPR.

```
521               <default>0</default>
522               <pChoices editable="false">
523                 <pChoice>0</pChoice>
524                 <pChoice>1</pChoice>
525                 <pChoice>*</pChoice>
526               </pChoices>
527               <help>Flag to match packets with IP options, * is wildcard</help>
528             </field>
529             <field editable="true">
530               <plabel>exception ttl</plabel>
531               <ptype>int</ptype>
532               <wildcard>-1</wildcard>
533               <default>0</default>
534               <pChoices editable="false">
535                 <pChoice>0</pChoice>
536                 <pChoice>1</pChoice>
537                 <pChoice>*</pChoice>
538               </pChoices>
539               <help>Flag to match packets with TTL=1,TTL=0, * is wildcard</help>
540             </field>
541             <field editable="true">
542               <plabel>tcp fin</plabel>
543               <ptype>int</ptype>
544               <wildcard>-1</wildcard>
545               <default>*</default>
546               <pChoices editable="false">
547                 <pChoice>0</pChoice>
548                 <pChoice>1</pChoice>
549                 <pChoice>*</pChoice>
550               </pChoices>
551               <help>Flag to match TCP FIN packets, * is wildcard</help>
552             </field>
553             <field editable="true">
554               <plabel>tcp syn</plabel>
555               <ptype>int</ptype>
556               <wildcard>-1</wildcard>
557               <default>*</default>
558               <pChoices editable="false">
559                 <pChoice>0</pChoice>
560                 <pChoice>1</pChoice>
561                 <pChoice>*</pChoice>
562               </pChoices>
563               <help>Flag to match TCP SYN packets, * is wildcard</help>
564             </field>
565             <field editable="true">
566               <plabel>tcp rst</plabel>
567               <ptype>int</ptype>
568               <wildcard>-1</wildcard>
569               <default>*</default>
570               <pChoices editable="false">
571                 <pChoice>0</pChoice>
572                 <pChoice>1</pChoice>
```

Listing B.1: (k) NPR.shw; specialization description for the NPR.

```
573                 <pChoice>*</pChoice>
574               </pChoices>
575               <help>Flag to match TCP RST packets, * is wildcard</help>
576             </field>
577             <field editable="true">
578               <plabel>tcp psh</plabel>
579               <ptype>int</ptype>
580               <wildcard>-1</wildcard>
581               <default>*</default>
582               <pChoices editable="false">
583                 <pChoice>0</pChoice>
584                 <pChoice>1</pChoice>
585                 <pChoice>*</pChoice>
586               </pChoices>
587               <help>Flag to match TCP PSH packets, * is wildcard</help>
588             </field>
589             <field editable="true">
590               <plabel>tcp ack</plabel>
591               <ptype>int</ptype>
592               <wildcard>-1</wildcard>
593               <default>*</default>
594               <pChoices editable="false">
595                 <pChoice>0</pChoice>
596                 <pChoice>1</pChoice>
597                 <pChoice>*</pChoice>
598               </pChoices>
599               <help>Flag to match TCP ACK packets, * is wildcard</help>
600             </field>
601             <field editable="true">
602               <plabel>tcp urg</plabel>
603               <ptype>int</ptype>
604               <wildcard>-1</wildcard>
605               <default>*</default>
606               <pChoices editable="false">
607                 <pChoice>0</pChoice>
608                 <pChoice>1</pChoice>
609                 <pChoice>*</pChoice>
610               </pChoices>
611               <help>Flag to match TCP URG packets, * is wildcard</help>
612             </field>
613             <field editable="true">
614               <plabel>qid</plabel>
615               <ptype>int</ptype>
616               <default>0</default>
617               <help>datagram queues(0-63) reserved queues(64-8191)</help>
618             </field>
619             <field editable="true">
620               <plabel>statsIndex</plabel>
621               <ptype>int</ptype>
622               <hwAssigned>statsIndex</hwAssigned>
623               <help>statistics index for monitoring matching flows</help>
624             </field>
```

Listing B.1: (l) NPR.shw; specialization description for the NPR.

```
625             <field editable="true">
626               <plabel>multicast</plabel>
627               <ptype>boolean</ptype>
628               <default>false</default>
629               <help>IPv4 multicast filter (makes multiple copies)</help>
630             </field>
631             <field editable="true">
632               <plabel>port_plugin_selection</plabel>
633               <ptype>string</ptype>
634               <default>port(unicast)</default>
635               <pChoices editable="false">
636                 <pChoice>port(unicast)</pChoice>
637                 <pChoice>plugin(unicast)</pChoice>
638                 <pChoice>ports and plugins (multicast)</pChoice>
639                 <pChoice>plugins(multicast)</pChoice>
640               </pChoices>
641               <help>determines how matching packets are forwarded to either
642                     a plugin or directly to an output port for unicast filters,
643                     and to only plugins or ports and plugins simultaneously
644                     for multicast filters</help>
645             </field>
646             <field editable="true">
647               <plabel>drop</plabel>
648               <ptype>boolean</ptype>
649               <default>false</default>
650               <help>check this to drop all matching packets</help>
651             </field>
652             <field editable="true">
653               <plabel>output_ports</plabel>
654               <ptype>string</ptype>
655               <default>0</default>
656               <pChoices editable="true">
657                 <pChoice>0</pChoice>
658                 <pChoice>1</pChoice>
659                 <pChoice>2</pChoice>
660                 <pChoice>3</pChoice>
661                 <pChoice>4</pChoice>
662                 <pChoice>use route</pChoice>
663               </pChoices>
664               <help>comma separated list of ports for multicast</help>
665             </field>
666             <field editable="true">
667               <plabel>output_plugins</plabel>
668               <ptype>string</ptype>
669               <default>0</default>
670               <pChoices editable="true">
671                 <pChoice>0</pChoice>
672                 <pChoice>1</pChoice>
673                 <pChoice>2</pChoice>
674                 <pChoice>3</pChoice>
675                 <pChoice>4</pChoice>
676               </pChoices>
```

Listing B.1: (m) NPR.shw; specialization description for the NPR.

```
677               <help>comma separated list of microengines for multicast</help>
678             </field>
679             <field editable="true" width="150">
680               <plabel>sampling_type</plabel>
681               <ptype>int</ptype>
682               <default>0</default>
683               <pChoices editable="false">
684                 <pChoice>0</pChoice>
685                 <pChoice>1</pChoice>
686                 <pChoice>2</pChoice>
687                 <pChoice>3</pChoice>
688               </pChoices>
689               <help>sampling type 0 samples 100%. 1, 2, and 3 percentages are
690                     set via top level NPR menu.</help>
691             </field>
692             <field editable="true" width="50">
693               <plabel>priority</plabel>
694               <ptype>int</ptype>
695               <default>50</default>
696               <help>lower numbered priorities will match packets first</help>
697             </field>
698           </fields>
699           <commands>
700             <addCommand opcode="76" numParams="28">
701               <displayLabel>AddFilter</displayLabel>
702               <description>add filter</description>
703               <param editable="true">
704                 <plabel>aux</plabel>
705                 <ptype>field</ptype>
706               </param>
707               <group>
708                 <param editable="true" width="150">
709                   <plabel>destination_address</plabel>
710                   <ptype>field</ptype>
711                 </param>
712                 <param editable="true">
713                   <plabel>destination_mask</plabel>
714                   <ptype>field</ptype>
715                 </param>
716               </group>
717               <group>
718                 <param editable="true" width="150">
719                   <plabel>source_address</plabel>
720                   <ptype>field</ptype>
721                 </param>
722                 <param editable="true">
723                   <plabel>source_mask</plabel>
724                   <ptype>field</ptype>
725                 </param>
726               </group>
727               <param editable="true">
728                 <plabel>plugin_tag</plabel>
```

Listing B.1: (n) NPR.shw; specialization description for the NPR.

```
729                 <ptype>field</ptype>
730               </param>
731               <param editable="true">
732                 <plabel>protocol</plabel>
733                 <ptype>field</ptype>
734               </param>
735               <param editable="true">
736                 <plabel>destination_port</plabel>
737                 <ptype>field</ptype>
738               </param>
739               <param editable="true">
740                 <plabel>source_port</plabel>
741                 <ptype>field</ptype>
742               </param>
743               <group>
744                 <param editable="true">
745                   <plabel>exception nonip</plabel>
746                   <ptype>field</ptype>
747                 </param>
748                 <param editable="true">
749                   <plabel>exception arp</plabel>
750                   <ptype>field</ptype>
751                 </param>
752                 <param editable="true">
753                   <plabel>exception ipopt</plabel>
754                   <ptype>field</ptype>
755                 </param>
756                 <param editable="true">
757                   <plabel>exception ttl</plabel>
758                   <ptype>field</ptype>
759                 </param>
760               </group>
761               <group label="tcp flags   ">
762                 <param editable="true">
763                   <plabel>tcp fin</plabel>
764                   <ptype>field</ptype>
765                 </param>
766                 <param editable="true">
767                   <plabel>tcp syn</plabel>
768                   <ptype>field</ptype>
769                 </param>
770                 <param editable="true">
771                   <plabel>tcp rst</plabel>
772                   <ptype>field</ptype>
773                 </param>
774                 <param editable="true">
775                   <plabel>tcp psh</plabel>
776                   <ptype>field</ptype>
777                 </param>
778                 <param editable="true">
779                   <plabel>tcp ack</plabel>
780                   <ptype>field</ptype>
```

Listing B.1: (o) NPR.shw; specialization description for the NPR.

```
781                 </param>
782                 <param editable="true">
783                   <plabel>tcp urg</plabel>
784                   <ptype>field</ptype>
785                 </param>
786               </group>
787               <param editable="true">
788                 <plabel>qid</plabel>
789                 <ptype>field</ptype>
790               </param>
791               <param editable="true">
792                 <plabel>statsIndex</plabel>
793                 <ptype>field</ptype>
794               </param>
795               <param editable="true">
796                 <plabel>multicast</plabel>
797                 <ptype>field</ptype>
798               </param>
799               <group>
800                 <param editable="true">
801                   <plabel>port_plugin_selection</plabel>
802                   <ptype>field</ptype>
803                 </param>
804                 <param editable="true">
805                   <plabel>drop</plabel>
806                   <ptype>field</ptype>
807                 </param>
808               </group>
809               <group>
810                 <param editable="true">
811                   <plabel>output_ports</plabel>
812                   <ptype>field</ptype>
813                 </param>
814                 <param editable="true">
815                   <plabel>output_plugins</plabel>
816                   <ptype>field</ptype>
817                 </param>
818               </group>
819               <group>
820                 <param editable="true">
821                   <plabel>sampling_type</plabel>
822                   <ptype>field</ptype>
823                 </param>
824                 <param editable="true">
825                   <plabel>priority</plabel>
826                   <ptype>field</ptype>
827                 </param>
828               </group>
829             </addCommand>
830             <deleteCommand opcode="77" numParams="19">
831               <displayLabel>DeleteFilter</displayLabel>
832               <description>delete filter</description>
```

Listing B.1: (p) NPR.shw; specialization description for the NPR.

```
            <param editable="false">
              <plabel>aux</plabel>
              <ptype>field</ptype>
            </param>
            <group>
              <param editable="false" width="150">
                <plabel>destination_address</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>destination_mask</plabel>
                <ptype>field</ptype>
              </param>
            </group>
            <group>
              <param editable="false" width="150">
                <plabel>source_address</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>source_mask</plabel>
                <ptype>field</ptype>
              </param>
            </group>
            <param editable="false">
              <plabel>plugin_tag</plabel>
              <ptype>field</ptype>
            </param>
            <param editable="false">
              <plabel>protocol</plabel>
              <ptype>field</ptype>
            </param>
            <param editable="false">
              <plabel>destination_port</plabel>
              <ptype>field</ptype>
            </param>
            <param editable="false">
              <plabel>source_port</plabel>
              <ptype>field</ptype>
            </param>
            <group>
              <param editable="false">
                <plabel>exception nonip</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>exception arp</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="false">
                <plabel>exception ipopt</plabel>
                <ptype>field</ptype>
```

Listing B.1: (q) NPR.shw; specialization description for the NPR.

```
                </param>
                <param editable="false">
                  <plabel>exception ttl</plabel>
                  <ptype>field</ptype>
                </param>
              </group>
              <group>
                <param editable="false">
                  <plabel>tcp fin</plabel>
                  <ptype>field</ptype>
                </param>
                <param editable="false">
                  <plabel>tcp syn</plabel>
                  <ptype>field</ptype>
                </param>
                <param editable="false">
                  <plabel>tcp rst</plabel>
                  <ptype>field</ptype>
                </param>
                <param editable="false">
                  <plabel>tcp psh</plabel>
                  <ptype>field</ptype>
                </param>
                <param editable="false">
                  <plabel>tcp ack</plabel>
                  <ptype>field</ptype>
                </param>
                <param editable="false">
                  <plabel>tcp urg</plabel>
                  <ptype>field</ptype>
                </param>
              </group>
            </deleteCommand>
          </commands>
          <monitoring>
            <command opcode="101" numParams="1">
              <displayLabel>StatsPreQPkt</displayLabel>
              <description>pre queue packet count for stats index</description>
              <units>packets</units>
              <param editable="false">
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
            <command opcode="102" numParams="1">
              <displayLabel>StatsPostQPkt</displayLabel>
              <description>post queue packet count for stats index</description>
              <units>packets</units>
              <param editable="false">
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
```

Listing B.1: (r) NPR.shw; specialization description for the NPR.

```
            </command>
            <command opcode="103" numParams="1">
              <displayLabel>StatsPreQByte</displayLabel>
              <description>pre queue byte count for stats index</description>
              <units>bytes</units>
              <param>
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
            <command opcode="104" numParams="1">
              <displayLabel>StatsPostQByte</displayLabel>
              <description>post queue byte count for stats index</description>
              <units>bytes</units>
              <param editable="false">
                <plabel>statsIndex</plabel>
                <ptype>field</ptype>
              </param>
            </command>
          </monitoring>
          <display>
            <column title="priority" width="50">
              <fieldName>priority</fieldName>
            </column>
            <column title="type" width="100">
              <symbol>aux:</symbol>
              <fieldName>aux</fieldName>
              <newline/>
              <symbol> sampling type:</symbol>
              <fieldName>sampling_type</fieldName>
              <newline/>
              <symbol>multicast:</symbol>
              <fieldName>multicast</fieldName>
            </column>
            <column title="address/mask port" width="200">
              <symbol>src:</symbol>
              <fieldName>source_address</fieldName>
              <symbol>/</symbol>
              <fieldName>source_mask</fieldName>
              <symbol> port </symbol>
              <fieldName>source_port</fieldName>
              <newline/>
              <symbol>dest:</symbol>
              <fieldName>destination_address</fieldName>
              <symbol>/</symbol>
              <fieldName>destination_mask</fieldName>
              <symbol> port </symbol>
              <fieldName>destination_port</fieldName>
            </column>
            <column title="protocol" width="53">
              <fieldName>protocol</fieldName>
            </column>
```

Listing B.1: (s) NPR.shw; specialization description for the NPR.

```
            <column title="tcpflags" width="55">
              <symbol>fin:</symbol>
              <fieldName>tcp fin</fieldName>
              <newline/>
              <symbol>syn:</symbol>
              <fieldName>tcp syn</fieldName>
              <newline/>
              <symbol>rst:</symbol>
              <fieldName>tcp rst</fieldName>
              <newline/>
              <symbol>psh:</symbol>
              <fieldName>tcp psh</fieldName>
              <newline/>
              <symbol>ack:</symbol>
              <fieldName>tcp ack</fieldName>
              <newline/>
              <symbol>urg:</symbol>
              <fieldName>tcp urg</fieldName>
              <newline/>
            </column>
            <column title="exceptions" width="65">
              <symbol>nonip:</symbol>
              <fieldName>exception nonip</fieldName>
              <newline/>
              <symbol>arp:</symbol>
              <fieldName>exception arp</fieldName>
              <newline/>
              <symbol>ipopt:</symbol>
              <fieldName>exception ipopt</fieldName>
              <newline/>
              <symbol>ttl:</symbol>
              <fieldName>exception ttl</fieldName>
              <newline/>
            </column>
            <column title="plugin tag" width="55">
              <fieldName>plugin_tag</fieldName>
            </column>
            <column title="output" width="100">
              <symbol>dropped:</symbol>
              <fieldName>drop</fieldName>
              <newline/>
              <symbol>ports:</symbol>
              <fieldName>output_ports</fieldName>
              <newline/>
              <symbol>plugins:</symbol>
              <fieldName>output_plugins</fieldName>
              <newline/>
              <symbol>pluginSelection:</symbol>
              <fieldName>port_plugin_selection</fieldName>
            </column>
            <column title="qid" width="50">
              <fieldName>qid</fieldName>
```

Listing B.1: (t) NPR.shw; specialization description for the NPR.

```
            </column>
            <column title="stats" width="50">
              <fieldName>statsIndex</fieldName>
            </column>
          </display>
        </entry>
      </table>
      <table title="QueueTable">
        <fields>
          <field editable="true">
            <plabel>bandwidth</plabel>
            <ptype>int</ptype>
            <default>1000</default>
            <updateCommand opcode="81">
              <displayLabel>SetPortRate</displayLabel>
              <description>set port rate</description>
            </updateCommand>
          </field>
        </fields>
        <commands>
        </commands>
        <monitoring>
        </monitoring>
        <entry name="queue" numFields="3" numColumns="3" enable="false">
          <fields>
            <field editable="true">
              <plabel>queue_id</plabel>
              <ptype>int</ptype>
              <default>0</default>
            </field>
            <field editable="true">
              <plabel>threshold</plabel>
              <ptype>int</ptype>
              <default>32768</default>
            </field>
            <field editable="true">
              <plabel>quantum</plabel>
              <ptype>int</ptype>
              <default>346</default>
            </field>
          </fields>
          <commands>
            <addCommand opcode="78" numParams="3">
              <displayLabel>ChangeQueue</displayLabel>
              <description>change queue parameters</description>
              <param editable="true">
                <plabel>queue_id</plabel>
                <ptype>field</ptype>
              </param>
              <param editable="true">
                <plabel>threshold</plabel>
                <ptype>field</ptype>
```

Listing B.1: (u) NPR.shw; specialization description for the NPR.

```
              </param>
              <param editable="true">
                <plabel>quantum</plabel>
                <ptype>field</ptype>
              </param>
            </addCommand>
          </commands>
          <monitoring>
            <command opcode="68" numParams="1">
              <displayLabel>ReadQueueLength</displayLabel>
              <description>read queue length</description>
              <units>bytes</units>
              <param editable="false">
                <plabel>queue_id</plabel>
                <ptype>field</ptype>
              </param>
            </command>
          </monitoring>
          <display>
            <column title="queue id" width="55">
              <fieldName>queue_id</fieldName>
            </column>
            <column title="threshold(bytes)" width="115">
              <fieldName>threshold</fieldName>
            </column>
            <column title="quantum" width="70">
              <fieldName>quantum</fieldName>
            </column>
          </display>
        </entry>
      </table>
    </tables>
  </ports>
</subtype>
```

Listing B.1: (v) NPR.shw; specialization description for the NPR.

References

[1] Matthew Adiletta, Mark Rosenbluth, Debra Bernstein, Gilbert Wolrich, and Hugh Wilkinson. The next generation of Intel IXP network processors. *Intel Technology Journal*, 6, 2002.

[2] Amazon. Elastic Compute Cloud website. `http://aws.amazon.com/ec2/`.

[3] David G. Andersen. Theoretical approaches to node assignment. 2002.

[4] David G. Andersen, Hari Balakrishnan, M. Frans Kaashoek, and Robert Morris. Experience with an evolving overlay network testbed. *SIGCOMM Comput. Commun. Rev.*, 33(3):13–19, 2003.

[5] Katerina Argyraki, Salman Baset, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Eddie Kohler, Maziar Manesh, Sergiu Nedevschi, and Sylvia Ratnasamy. Can software routers scale? In *PRESTO '08: Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, pages 21–26, New York, NY, USA, 2008. ACM.

[6] Arista. Arista 7100 series switches. `http://www.aristanetworks.com/en/7100Series`.

[7] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. Above the clouds: A Berkeley view of cloud computing. Technical report, University of California, Berkeley, 2009.

[8] Yossi Azar and Oded Regev. Strongly polynomial algorithms for the unsplittable flow problem. In *In Proceedings of the 8th Conference on Integer Programming and Combinatorial Optimization (IPCO)*, pages 15–29, 2001.

[9] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI veritas: Realistic and controlled network experimentation. In *SIGCOMM '06: Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 3–14, New York, NY, USA, 2006. ACM.

[10] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS '07: Proceedings of the 3rd ACM/IEEE*

Symposium on Architecture for networking and communications systems, pages 145–154, New York, NY, USA, 2007. ACM.

[11] Terry Benzel, Robert Braden, Dongho Kim, Clifford Neuman, Anthony Joseph, Keith Sklower, Ron Ostrenga, and Stephen Schwab. Design, deployment, and use of the DETER testbed. In *DETER: Proceedings of the DETER Community Workshop on Cyber Security Experimentation*, pages 1–1, Berkeley, CA, USA, 2007. USENIX Association.

[12] Justin Cappos, Ivan Beschastnikh, Arvind Krishnamurthy, and Tom Anderson. Seattle: A platform for educational cloud computing. In *SIGCSE '09: Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, pages 111–115, New York, NY, USA, 2009. ACM.

[13] Amit Chakrabarti, Chandra Chekuri, Anupam Gupta, and Amit Kumar. Approximation algorithms for the unsplittable flow problem. In *APPROX '02: Proceedings of the 5th International Workshop on Approximation Algorithms for Combinatorial Optimization*, pages 51–66, London, UK, 2002. Springer-Verlag.

[14] Michael K. Chen, Xiao Feng Li, Ruiqi Lian, Jason H. Lin, Lixia Liu, Tao Liu, and Roy Ju. Shangri-La: achieving high performance from compiled network applications while enabling ease of programming. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '05, pages 224–236, New York, NY, USA, 2005. ACM.

[15] Sumi Choi, John Dehart, Anshul Kantawala, Ralph Keller, Fred Kuhns, John Lockwood, Prashanth Pappu, Jyoti Parwatikar, W. David Richard, Ed Spitznagel, David Taylor, Jonathan Turner, and Ken Wong. Design of a high performance dynamically extensible router. In *Proceedings of the DARPA Active Networks Conference and Exposition*, May 2002.

[16] Cisco. IOS website. `http://www.cisco.com/en/US/products/sw/iosswrel/products_ios_cisco_ios_software_category_home.html`.

[17] Jeffrey Considine, John W. Byers, and Ketan Meyer-Patel. A constraint satisfaction approach to testbed embedding services. *SIGCOMM Comput. Commun. Rev.*, 34(1):137–142, 2004.

[18] World Wide Web Consortium. Extensible markup language (XML). `http://www.w3.org/XML/`.

[19] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, second edition, 2001.

[20] G. Adam Covington, Glen Gibb, John Lockwood, and Nick McKeown. A packet generator on the NetFPGA platform. In *The 17th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 5–7 April 2009.

[21] Dan Decasper, Zubin Dittia, Guru Parulkar, and Bernhard Plattner. Router plugins: a software architecture for next generation routers. In *Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication*, SIGCOMM '98, pages 229–240, New York, NY, USA, 1998. ACM.

[22] John DeHart, Fred Kuhns, Jyoti Parwatikar, Jonathan Turner, Charlie Wiseman, and Ken Wong. The Open Network Laboratory. In *SIGCSE '06: Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, pages 107–111, New York, NY, USA, 2006. ACM.

[23] Tasos Dimitriou, John Kolokouris, and Nikos Zarokostas. Sensenet: A wireless sensor network testbed. In *MSWiM '07: Proceedings of the 10th ACM Symposium on Modeling, Analysis, and Simulation of Wireless and Mobile Systems*, pages 143–150, New York, NY, USA, 2007. ACM.

[24] Marcel Dischinger, Andreas Haeberlen, Ivan Beschastnikh, Krishna P. Gummadi, and Stefan Saroiu. Satellitelab: Adding heterogeneity to planetary-scale network testbeds. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, pages 315–326, New York, NY, USA, 2008. ACM.

[25] FSArchiver. File system archiver for linux. `http://www.fsarchiver.org`.

[26] GENI. Global Environment for Network Innovations website. `http://www.geni.net`.

[27] G. Gibb, J. W. Lockwood, J. Naous, P. Hartke, and N. McKeown. NetFPGA: Open platform for teaching how to build gigabit-rate network switches and routers. 51(3):364–369, Aug. 2008.

[28] Google. AppEngine website. `http://code.google.com/appengine/`.

[29] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A clean slate 4d approach to network control and management. *SIGCOMM Comput. Commun. Rev.*, 35(5):41–54, 2005.

[30] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an operating system for networks. *SIGCOMM Comput. Commun. Rev.*, 38(3):105–110, 2008.

[31] Gurobi. Gurobi website. `http://www.gurobi.com`.

[32] Mart Haitjema, Ritun Patney, Jon Turner, Charlie Wiseman, and John DeHart. Performance-engineered network overlays for high quality interaction in virtual

worlds. Technical Report WUCSE-2009-18, Washington University in St. Louis, June 2009.

[33] Mark Handley, Eddie Kohler, Atanu Ghosh, Orion Hodson, and Pavlin Radoslavov. Designing extensible IP router software. In *Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 189–202, Berkeley, CA, USA, 2005. USENIX Association.

[34] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. TWIST: A scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *REALMAN '06: Proceedings of the 2nd International Workshop on Multi-hop Ad hoc Networks: From Theory to Reality*, pages 63–70, New York, NY, USA, 2006. ACM.

[35] Intel. Intel nehalem microarchitecture. `http://www.intel.com/technology/architecture-silicon/next-gen`.

[36] S. James and C. Crowley. IMP: ISP-Managed P2P. In *Peer-to-Peer Computing, 2010. P2P '10. IEEE Tenth International Conference on*, 2010.

[37] D. Johnson, T. Stack, R. Fish, D. M. Flickinger, L. Stoller, R. Ricci, and J. Lepreau. Mobile Emulab: A robotic wireless and sensor network testbed. In *Proc. 25th IEEE International Conference on Computer Communications INFOCOM 2006*, pages 1–12, April 2006.

[38] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The Click modular router. *ACM Trans. Comput. Syst.*, 18(3):263–297, 2000.

[39] Petr Kolman and Christian Scheideler. Improved bounds for the unsplittable flow problem. volume 61, pages 20–44, Duluth, MN, USA, 2006. Academic Press, Inc.

[40] Siming Lin, Xueqi Cheng, and Jianming Lv. A visualized parallel network simulator for modeling large-scale distributed applications. In *Proc. Eighth International Conference on Parallel and Distributed Computing, Applications and Technologies PDCAT '07*, pages 339–346, December 3–6, 2007.

[41] J. W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and Jianying Luo. NetFPGA–an open platform for gigabit-rate network switching and routing. In *Proc. IEEE International Conference on Microelectronic Systems Education MSE '07*, pages 160–161, 3–4 June 2007.

[42] Jing Lu and Jonathan Turner. Efficient mapping of virtual networks onto a shared substrate. Technical Report WUCSE-2006-35, Washington University, June 2006.

[43] Jad Naous, David Erickson, G. Adam Covington, Guido Appenzeller, and Nick McKeown. Implementing an OpenFlow switch on the NetFPGA platform. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 1–9, New York, NY, USA, 2008. ACM.

[44] Jad Naous, Glen Gibb, Sara Bolouki, and Nick McKeown. NetFPGA: Reusable router architecture for experimental research. In *PRESTO '08: Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, pages 1–7, New York, NY, USA, 2008. ACM.

[45] Netgear. GSM7532S website. `http://netgear.com/Products/Switches/FullyManaged10_100_1000Switches/GSM7352S.aspx`.

[46] James Newsome and Dawn Song. GEM: Graph embedding for routing and data-centric storage in sensor networks without geographic information. In *SenSys '03: Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 76–88, New York, NY, USA, 2003. ACM.

[47] NFS. Network file system. `http://nfs.sourceforge.net`.

[48] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. PortLand: A scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, pages 39–50, New York, NY, USA, 2009. ACM.

[49] NIS. Network information service. `http://www.linux-nis.org`.

[50] ns 2. ns-2 network simulator website. `http://www.isi.edu/nsnam/ns/`.

[51] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The Eucalyptus open-source cloud-computing system. In *Proc. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid CCGRID '09*, pages 124–131, May 18–21, 2009.

[52] OpenNMS. OpenNMS website. `http://www.opennms.org`.

[53] OpenSSH. OpenSSH website. `http://www.openssh.org`.

[54] David Oppenheimer, Jeannie Albrecht, David Patterson, and Amin Vahdat. Distributed resource discovery on PlanetLab with SWORD. In *WORLDS '04: Proceedings of the First Workshop on Real, Large Distributed Systems*, December 2004.

[55] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A blueprint for introducing disruptive technology into the Internet. In *Proceedings of HotNets–I*, Princeton, New Jersey, October 2002.

[56] Radisys. Radisys ATCA products website. `http://www.radisys.com/Products/ATCA.html`.

[57] D. Raychaudhuri, I. Seskar, M. Ott, S. Ganu, K. Ramachandran, H. Kremo, R. Siracusa, H. Liu, and M. Singh. Overview of the ORBIT radio grid testbed for evaluation of next-generation wireless network protocols. In *Proc. IEEE Wireless Communications and Networking Conference*, volume 3, pages 1664–1669, 13–17 March 2005.

[58] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *SIGCOMM Comput. Commun. Rev.*, 33(2):65–81, 2003.

[59] Robert Ricci, David Oppenheimer, Jay Lepreau, and Amin Vahdat. Lessons from resource allocators for large-scale multiuser testbeds. *SIGOPS Oper. Syst. Rev.*, 40(1):25–32, 2006.

[60] George F. Riley, Mostafa H. Ammar, Richard M. Fujimoto, Alfred Park, Kalyan Perumalla, and Donghua Xu. A federated approach to distributed network simulation. *ACM Trans. Model. Comput. Simul.*, 14(2):116–148, 2004.

[61] Niraj Shah, William Plishker, and Kurt Keutzer. NP-Click: A programming model for the Intel IXP1200. In *In 2nd Workshop on Network Processors (NP-2) at the 9th International Symposium on High Performance Computer Archtecture (HPCA-9)*, pages 100–111. Morgan Kaufmann, 2003.

[62] U. Shevade, R. Kokku, and H. M. Vin. Run-time system for scalable network services. In *Proc. INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 1813–1821, April 13–18, 2008.

[63] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a robust software-based router using network processors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*, SOSP '01, pages 216–229, New York, NY, USA, 2001. ACM.

[64] Synaccess. Synaccess NP-08G2. `http://www.synaccess-net.com/remote-power.php/1/12`.

[65] SSH Tunneling. Tunneling explained. `http://www.ssh.com/support/documentation/online/ssh/winhelp/32/Tunneling_Explained.html`.

[66] Jonathan S. Turner, Patrick Crowley, John DeHart, Amy Freestone, Brandon Heller, Fred Kuhns, Sailesh Kumar, John Lockwood, Jing Lu, Michael Wilson,

Charles Wiseman, and David Zar. Supercharging PlanetLab: A high performance, multi-application, overlay network platform. In *SIGCOMM '07: Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 85–96, New York, NY, USA, 2007. ACM.

[67] WAIL. Wisconsin Advanced Internet Laboratory website. `http://www.schooner.wail.wisc.edu`.

[68] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. MoteLab: A wireless sensor network testbed. In *IPSN '05: Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, page 68, Piscataway, NJ, USA, 2005. IEEE Press.

[69] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI '02: Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, pages 255–270, New York, NY, USA, 2002. ACM.

[70] Charlie Wiseman, Jyoti Parwatikar, Ken Wong, John DeHart, and Jonathan Turner. Design of an extensible network testbed. In *NCA '10: Proceedings of the 2010 Ninth IEEE International Symposium on Network Computing and Applications*, 2010.

[71] Charlie Wiseman, Jonathan Turner, Michela Becchi, Patrick Crowley, John DeHart, Mart Haitjema, Shakir James, Fred Kuhns, Jing Lu, Jyoti Parwatikar, Ritun Patney, Michael Wilson, Ken Wong, and David Zar. A remotely accessible network processor-based router for network experimentation. In *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 20–29, New York, NY, USA, 2008. ACM.

[72] Charlie Wiseman, Jonathan Turner, and Patrick Crowley. The Open Network Laboratory. In *SIGCOMM '09: Proceedings of the 2009 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*. ACM, 2009.

[73] Charlie Wiseman, Jonathan Turner, John DeHart, Jyoti Parwatikar, Ken Wong, and David Zar. Using the NetFPGA in the Open Network Laboratory. In *Proceedings of the 1st NetFPGA DevelopersWorkshop*, 2009.

[74] Charlie Wiseman, Ken Wong, Tilman Wolf, and Sergey Gorinsky. Operational experience with a virtual networking laboratory. In *SIGCSE '08: Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education*, pages 427–431, New York, NY, USA, 2008. ACM.

[75] T. Wolf. Assessing student learning in a virtual laboratory environment. *Education, IEEE Transactions on*, 53(2):216 –222, May 2010.

[76] Ken Wong, Tilman Wolf, Sergey Gorinsky, and Jonathan Turner. Teaching experiences with a virtual network laboratory. In *SIGCSE '07: Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, pages 481–485, New York, NY, USA, 2007.

[77] Xilinx. Xilinx website. `http://www.xilinux.com`.

[78] Fang Yu, Zhifeng Chen, Yanlei Diao, T. V. Lakshman, and Randy H. Katz. Fast and memory-efficient regular expression matching for deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 93–102, New York, NY, USA, 2006. ACM.

[79] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *SIGCOMM Comput. Commun. Rev.*, 38(2):17–29, 2008.

[80] Y. Zhu and M. Ammar. Algorithms for assigning substrate network resources to virtual network components. In *Proc. 25th IEEE International Conference on Computer Communications INFOCOM 2006*, pages 1–12, April 2006.