UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Network Control Platform for Performance Isolation and Modular Composition

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Kevin Christopher Webb

2013

The Dissertation of Kevin Christopher Webb is approved and is acceptable in quality and form for publication on microfilm and electronically:

Co-Chair

Co-Chair

University of California, San Diego

2013

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I am deeply grateful to my advisors, Alex Snoeren and Ken Yocum, for their advice and feedback on paper drafts, slides, graphs, job application materials, practice talks, and countless other things. Their encouragement and guidance always push me to improve and steer my focus towards the right piece of the puzzle, even when I haven't realized its importance yet.

I would like to express my gratitude to Beth Simon and Joe Pasquale for demonstrating excellence and passion for teaching computer science. They have inspired me to pursue, and helped me to obtain, an amazing academic opportunity.

I want to thank my friends and fellow students from UCSD, Georgia Tech, and Farragut for celebrating my sucesses, taking the sting out of paper rejections, and providing moral support along this journey.

Finally, my heartfelt appreciation goes to my wife Melissann and my parents, Bill and Pat. Without their love, support, and encouragement, I could never have made it this far!

Chapters 3 and 4 contain material as it appears in the Proceedings of the USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE) 2011. "Topology Switching for Data Center Networks". Webb, Kevin; Snoeren, Alex; Yocum, Ken. The dissertation author was the primary investigator and author of this paper.

Chapters 2, 3, and 5 contain material that has been submitted for publication in the Proceedings of the ACM Symposium on Cloud Computing (SOCC), 2013. "Blender: Mixing Data Center Network Isolation Models". Webb, Kevin; Yocum, Ken; Snoeren, Alex. The dissertation author was the primary investigator and author of this paper.

Chapter 6 contains material that is currently being prepared for submission for publication. Webb, Kevin; Roy, Arjun; Yocum, Ken; Snoeren, Alex. The dissertation

author was the primary investigator and author of this material.

VITA

2007	Bachelor of Science Computer Science Georgia Institute of Technology
2010	Master of Science Computer Science University of California, San Diego
2013	Doctor of Philosophy Computer Science University of California, San Diego

ABSTRACT OF THE DISSERTATION

A Network Control Platform for Performance Isolation and Modular Composition

by

Kevin Christopher Webb

Doctor of Philosophy in Computer Science

University of California, San Diego, 2013

Alex Snoeren, Co-Chair
Ken Yocum, Co-Char

Large data centers host thousands of tenants and services, and today's data center networks are no longer simple end-to-end transfer fabrics. Tenants desire more functionality like strong performance isolation, latency control, load balancing, middlebox placement, and other services. As tenant performance requirements become more demanding, data center providers wish to increase the set of services they offer in an attempt to meet the demand and attract new customers. Critically, while the promise of cloud computing makes it straightforward to rent compute resources, there is no standard model for tenants to pick and choose the network resources and services they require.

Traditionally, a single procedure governs the allocation of resources to tenant services. For efficiency, data center providers multiplex tenants across the physical infrastructure, leveraging end host virtualization to carve out well-specified units of isolated CPU, memory, and storage. In contrast, tenants typically receive loose, qualitative descriptions of network performance with a limited set of available features. This disparity is problematic for today's data centers, which need to satisfy services with a diverse and non-uniform set of networking needs.

This dissertation proposes to resolve this contention by provisioning a custom virtual network for each of the data center's clients. It presents task switching, a framework that redefines how tenants interact with and obtain resources from the network by servicing their individual requests for virtual network *tasks*. It then describes a model with two components, Blender and Omakase, that defines interfaces for the network provider to implement differing features and resource allocation procedures that tenants may choose to adopt for their tasks.

Blender focuses on performance isolation, eliminating the either/or balancing act between resource efficiency and performance predictability by allowing data center operators to run multiple performance isolation models simultaneously. Omakase augments the Blender model to incorporate general purpose features for tasks, including customized in-network middlebox processing, failure resolution, and flow scheduling. We evaluate the architecture with a prototype implementation that addresses its correctness, scalability, and performance. We show that our SDN-based prototype can efficiently express many recently proposed performance isolation models and network features.

Chapter 1

Introduction

In recent years, large Internet-based companies have raced to construct ever-larger data centers to support Web search, e-mail, social networking, and other global Internet services. A "perfect storm" of recent trends has fueled data center enthusiasm, with several major innovations driving data center demand:

- The promise of infinitely available computing and storage in the cloud has motivated the construction of Amazon's EC2 [5] and Microsoft's Windows Azure [57] computing platforms. Traditional desktop applications like word processors and spreadsheets have migrated to the Web under the "software as a service" model.

- Large interactive and data-intensive social media services such as Facebook, Twitter, and Flickr have profoundly changed the way that users contact with friends and family, conduct business, and interact with governments. Such services support an enormous number of concurrent user requests and require substantial computing resources.

- Applications like MapReduce [22], Hadoop [29], and Dryad [43] provide powerful and readily available data processing architectures for large clusters of commodity machines.

At the same time, these companies are responding to economic incentives to build larger data centers to reduce their operational expenses. By establishing an economy of scale, they amortize the costs of managing infrastructure, cooling and powering servers, and utilizing expensive high-speed Internet communication links. Another economic trend in data centers is the movement towards “scale out rather than scale up” architectures, meaning that data center administrators often forgo investing in expensive, “enterprise-grade” server equipment and instead opt to purchase a larger quantity of cheaper commodity hardware. Because single-thread CPU performance [90] and single-link network performance [2] are plateauing, today’s data centers require tens to hundreds of thousands of commodity servers to meet their performance demands.

Combined, these trends place a considerable strain on a crucial component of the data center environment: the network. Despite being housed in a single physical structure, a data center’s network infrastructure is shared by multiple tenants (i.e., users who execute services and applications), and the architecture more closely resembles the Internet than a conventional local area network. Data center networks constitute a unique operating environment with two primary challenges:

- **Topology and Routing:** Conventional network topologies rely on hierarchical designs that require increasingly high-performance network hardware at higher tiers. Due to a lack of sufficiently powerful devices, these designs fail to supply adequate capacity at data center scales. Instead, contemporary data center networks exploit numerous parallel paths to achieve fault tolerance and meet performance demands [2]. Unfortunately, relying on an abundance of parallel links spreads the network’s resources across more devices and ports, which increases the complexity of assigning resources to tenants.

  As the level of redundancy increases, it becomes necessary to balance traffic over parallel links to fully utilize the network. In most data centers, Ethernet is the preferred physical-layer technology for cost and compatibility, but its protocols are designed for single spanning-tree environments where frequent broadcasting is acceptable. The increasingly common solution [16, 32] of adopting IP routing to divide the network into multiple subnetworks increases the cost of network devices and limits the maximum size of the network due to hardware routing table size limitations [62].

Data center networks must carefully control the placement of services and selection of paths to fully utilize the network without overflowing the limited capacity of switch and router forwarding tables.

- **Multi-tenancy:** To leverage their economies of scale, data center administrators place a diverse set of tenants across a shared network infrastructure. Sharing links is problematic for conventional networks, which treat traffic equally, expect an abundance of resources, and assume that the tenants sharing the network are not adversarial. To maximize the return on their infrastructure investment, data center administrators would prefer to differentiate between their tenants, customizing tenant network resource allocations according to their resource demands, their relative importance, and in the case of cloud computing, the amount that they pay for service.

  To effectively differentiate between tenants, the network must distinguish one tenant's traffic from another, determine a plan for allocating the network's resources to tenants, and enforce the plan's resource assignment. For example, suppose a cloud network provider is hosting two tenant *tasks*: a latency-sensitive Web service that needs to quickly respond to customer requests and a bandwidth-hungry bulk processing application. In this case, the network administrators (or a control platform running on their behalf) must answer: Which paths should each task be permitted to use? On permitted paths, how much of the available capacity should each task be assigned? How might the resource allocation change if a third task arrives?

Finally, separating tenant traffic exacerbates switch forwarding table capacity limitations. As the number of tenants increases, devices need multiple forwarding table entries to delineate tenants' tasks. Thus, data center networks need a control platform to automate scalable resource allocation and enforcement for a diverse collection of tenants.

Fortunately, the data center domain also represents a novel opportunity to forgo traditional network designs. Unlike the public Internet, which requires a coalition of support from standards committees and coordinated deployment from Internet service providers, data centers are typically owned by a single entity. Exclusive ownership frees the owner to make unilateral design decisions, allowing them to adopt unconventional network architectures, protocols, and control platforms.

Due to recent developments towards open, programmable network devices [55], network operators can rapidly develop and deploy new network protocols and management infrastructure. Network devices are beginning to resemble general-purpose platforms; they export standard abstractions, similar to end host hardware, with constructs like flows, tables, rate limiters, and queues rather than CPU registers, memory, etc. On end hosts, an operating system arbitrates access to these resources between applications. This dissertation explores a new network architecture for similarly abstracting and managing the network's resources in an environment shared by multiple tenants.

1.1 Hypothesis

Current network platforms take a tenant-agnostic approach to supporting data center applications. The hypothesis of this dissertation is that data center networks would improve tenant performance and enable critical management features by customizing the network's behavior according to tenant needs. Employing specialized control software would enable the network to independently provision resources and tailor a virtual network environment for each of its tenants. Furthermore, the tenants' services would execute more effectively by composing disparate features such as performance isolation, routing, failure recovery, and middlebox interposition.

This dissertation examines the design of a network resource allocation model with two components, Blender and Omakase, in the context of an architecture that tunes the network's features and performance to match the needs of its tenants. Blender focuses specifically on network isolation – preventing the applications of one tenant from interfering with those of another. Omakase extends Blender with a general model for augmenting the network with composable support functionality.

1.2 Contributions

This dissertation makes four contributions for multiplexing tenants over a shared data center network infrastructure:

- **Task Switching:** Task switching shields tenant applications from the details of the physical network. Rather than individual links or devices, task switching allocates resources to tenant applications as a whole, at the granularity of a *task*. Tasks need only make a high-level request for resources, and the network's centralized control software will handle the details. Task switching, detailed in Chapter 3, is the design principle underlying the model subsequently presented in this dissertation. We analyze the feasibility of task switching with simulations in Chapter 4.

- **Composable Performance Isolation (Blender):** Blender introduces a model for mixing data center network isolation primitives. Blender allows network operators to reserve resources on behalf of tasks while ensuring consistency across a shared physical network infrastructure. Chapter 5 describes Blender's modular, graph-based abstractions and demonstrates their ability to express desirable performance isolation strategies.

- **Network Functionality "App Store" (Omakase):** In Chapter 6, we augment Blender's abstractions with Omakase, a model for supporting dynamically executing applications within the network. Omakase defines a novel event processing and network location interface for building and composing general network functionality in support of network tasks.

- **Prototypes and Evaluation** To evaluate the Blender and Omakase abstractions, we construct an SDN-based prototype implementation. In our evaluation of Blender, we show that the prototype can express and extend proposed isolation models on existing SDN hardware. Our experiments quantify Blender's performance and potential for deployment at scale. For Omakase, our prototype demonstrates the effectiveness of several application classes, including dynamic path selection and deadline-aware flow scheduling.

Chapter 2

Background and Related Work

The architecture presented in this dissertation takes its inspiration from a number of previous projects. This chapter explores the relevant background material, related work, and their relationship to Blender and Omakase, beginning with a characterization of the data center operating environment.

2.1 Data Center Environment

As Chapter 1 described, many technology companies are rapidly increasing the number and capacity of new data center installations. To quantify the growth trends, a recent Facebook press release [65] describes the company's plan to construct a new 476,000-square foot data center in Altoona, Iowa – a project that represents a $300+ million capital investment. This announcement comes only two years after their completion of a 300,000-square foot data center in Prineville, Oregon [25]. Google operates 13 data centers worldwide [31], and Microsoft opened two new data centers in 2009 and one in 2011 [30], one of which was a 700,000-square foot, $500 million investment [58]. This section provides context for the remainder of this dissertation with a description of the actors, network device architecture, and virtualized end hosts commonly found in data centers.

2.1.1 Actors

Data center network architectures are rapidly evolving to accommodate the demands of the “cloud computing” model, in which data centers dynamically provision raw computing resources or application services on demand. The data center may be *public*, where the resources are rented out to anyone willing to pay, or *private*, supporting only the services of data center’s owner. In either case, we make a distinction between two primary actors: the data center provider, who owns and manages the data center’s resources, and service operators, who use those resources to support Web, social media, or other user-facing services[1]. Note that for a private data center, despite both actors working for the same organization towards a similar goal, we maintain the distinction between the provider (who controls the resources) and the service operator (who requests to use them). Further, we classify resources as residing at end hosts or within network devices (i.e. switches and routers).

2.1.2 End hosts

Data centers employ end hosts to execute their applications. To reduce costs, end hosts are generally servers that house commodity CPUs, memory, and disks. To meet the disparate needs of a diverse user base, data center providers virtualize server hardware, meaning the physical resources are partitioned such that differing services can utilize (and, if the data center is public, pay for) only the resources they request. On each server, virtual machine management software known as a hypervisor carves up the available physical resources and assigns them to one or more virtual machines, each of which executes its own operating system and software stack. Critically, these platforms enforce *performance isolation* between virtual machines – that is one virtual machine

[1]In the context of shared or public data center networks, we often refer to the data center provider as the network provider and to service operators as tenants.

will not negatively impact the performance of another, even when they share the same physical host. End-host virtualization is a mature technology, with multiple commercial products [67, 93] and open-source virtualization platforms [9, 48] available.

2.1.3 Network Hardware

Unlike end hosts, typical network devices are composed of domain-specific hardware divided into two functions, or *planes*: a data plane and a control plane. The data plane represents the packet forwarding path for the device. In today's devices, data plane hardware quickly classifies and forwards packets at line rates of 1, 10, or 40 Gbps. Devices generally perform classification with a ternary content addressable memory (TCAM) [68], which allows for parallel matching of packet headers that can optionally ignore sections of the header (via "don't care" patterns). Matched entries trigger an action, such as dropping the packet, forwarding it out a designated port, sending it to a particular queue, or otherwise determining the packet's fate. TCAMs are expensive and power-hungry, and thus most devices have constrained TCAM capacities that require careful management to avoid overflowing.

The control plane manages the data plane via a general purpose CPU over a relatively low-performance interconnect [21]. Network devices access the control plane infrequently, typically to reconfigure the data plane hardware in response to network topology changes (e.g., failures) or administrative configuration changes. Traditional network devices ship with vendor-supplied control plane software that allows for network provider configuration but offers little opportunity for provider programmability (i.e., the TCAM is not directly accessible by the network provider).

Only recently have things begun to change. Due to growing pressure from a new movement gaining traction in the networking community, vendors have begun to compromise in the form of software-defined networking (SDN). The premise of

SDN eschews conventional distributed protocols, opting instead to employ a logically-centralized network controller. Devices receive instructions directly from the controller, which assumes authority over a standardized, high-level management interface rather than directly manipulating hardware. In the leading SDN alternative, OpenFlow [55], the data plane only modifies the data plane's forwarding configuration in response to explicit controller directives called rules. Our prototype makes extensive use of OpenFlow's features, and future chapters describe mechanisms for rule compilation, optimization, and installation.

Critically, mature virtualization solutions do not exist for network devices. One of the primary goals of the architecture described in this dissertation is to provide a network control platform that enables similar virtualization functionality for data center network hardware. The remainder of this chapter examines approaches for realizing high throughput across redundant paths, followed by methods for virtualizing the network, control platforms to simplify network programming, and other systems that improve network virtualization functionality or efficiency.

2.2 Maximizing Throughput

Demand for increased capacity has outpaced the development of faster physical-layer standards, and as a result, data center providers are turning to highly redundant network topologies [1, 2, 35, 37, 87]. To be utilized effectively, the network must divide its traffic across many parallel paths. Some techniques are traffic oblivious, favoring simplicity and low overhead, while others collect detailed flow statistics in support of intricate flow schedules.

Traffic-oblivious methods include equal-cost multi-path routing (ECMP) [42], in which each device along a flow's path pins it to a random output port when multiple, equal-cost routing options are available. Another system, VL2 [32], combines ECMP with Valiant load balancing to cope with ECMP path count limitations and spread traffic more uniformly across a large set of path choices. SPAIN [61] creates multiple spanning trees, separated by virtual local-area networks (VLANs), to create multiple path choices. To spread traffic across redundant paths, end hosts randomly assign flows to VLANs, with periodic reassignment to avoid pathological flow collision scenarios. These traffic-oblivious multi-path routing schemes randomly spread flows across redundant paths, resulting in good average-case performance when flows are similarly sized, but their performance suffers when flow sizes are non-uniform [3].

Traffic-aware systems take flow size and endpoints into consideration to produce a global placement schedule. Hedera [3] estimates flow demand in the network's switches, and a central controller periodically reassigns flows to separate competing flows and reduce link contention. Mahout [20] also centrally computes a global schedule, but it collects flow information from end host socket buffers rather than network switches, reducing the overhead on the network.

Another class of traffic-aware systems strategically rearranges the network's physical capacity in response to link congestion. Helios [26] and c-Through [97] apply optics to increase the accessible bandwidth in targeted locations, relieving hot spots. Flyways [38] similarly augment networks with positionable high-speed wireless links.

In general, these performance-maximizing mechanisms are attractive to data center providers because they help the network, a significant investment, to operate at high overall efficiency. They perform best when the data center supports a single, bandwidth-hungry class of service, but they are ill-suited to multi-tenant environments where services may be throughput and latency sensitive [12, 13, 38, 46]. In that case, services benefit from provisioning resources on a per-tenant basis, at the expense of some link utilization efficiency. We explore this trade-off in our discussion of multi-tenancy.

2.3 Multi-tenancy

Due to the economic concerns outlined in Chapter 1 data center providers allow multiple service operators to share their physical network infrastructure. Sharing in networks is not a new phenomenon – statistical multiplexing is a historical principle in networking. However, data centers generally posses more stringent operational requirements than conventional local-area networks. Data center tenants often pay for a level of performance governed by a contract known as a service-level agreement (SLA) [70]. In particular, data center networks support an increasingly sophisticated environment that includes storage, monitoring, data processing, and virtual machine management software. These services place different performance, reliability, and management demands on the underlying network. For example, while caching services (e.g., memcached [27]) or HPC workloads prize low-latency communication, data processing applications (e.g., MapReduce [22]) require high bisection bandwidth.

The common approach to virtualizing traditional networks is to segregate traffic into multiple virtual local-area networks (VLANs), based on executing services or organizational units. Applying VLANs to the data center suffers from two key deficiencies:

1. VLAN separation fails to account for the diversity of tenant requirements. It provides only coarse control for assigning links to tenants, and the same underlying network protocols choose the “best” route for each tenant. For example, traditional Ethernet routes across a single spanning tree, resulting in non-shortest paths, idle alternative paths, and high link stress at the root of the network.

2. VLANs logically isolate tenant applications such that traffic is not visible between them, but it makes no guarantee about performance. Ensuring the network meets multiple SLAs in an environment where tenants are potentially adversarial (or

at least non-cooperative) requires performance isolation, which VLANs do not provide.

2.3.1 Performance Isolation

Network performance isolation is difficult to provide due to the distributed nature of network hardware devices, which unlike end hosts, require substantial coordination. The necessity of tenant isolation forces data center providers to balance network efficiency with performance guarantees for individual tenants. A large body of work has focused on identifying isolation mechanisms that provide an appropriate degree of multiplexing within this continuum, fundamentally trading off efficient utilization with predictable performance [72]. This trade-off is complex to navigate, as no single point in the spectrum appears to be suitable for all tenants.

We summarize several recent isolation systems in Table 2.1. To balance the conflicting needs of isolation and sharing, these proposals group and isolate traffic at granularities larger than individual flows. We categorize isolation approaches based upon this distinction: those that group traffic by entity (most commonly all or a subset of flows from a VM) and those that group by tenant (i.e., collection of VMs). We term these *entity* and *tenant*-based models, respectively. Table 2.1 further classifies each system based upon whether they provide absolute ("Fixed") or relative ("Proportional") performance isolation between tenants. Fixed allocations provide bandwidth guarantees, allowing the system to offer predictable performance at the expense of utilization.

Table 2.1. A summary of recently proposed data center network isolation models.

Isolation Unit	**System**	**Allocations**	**Work Conserving**	**Dynamic Coordination**	**Admission Control**	**Goal**	**Assumptions**
Tenant-based	Oktopus [8]	Fixed	No	Yes, DRL	Yes	Virtual tree-based network	Tree-based topologies
	GateKeeper [76]	Fixed	Yes	Yes	Yes	Virtual network	Bottleneck-free core
	NetShare [49]	Proportional	Yes	Yes	No	Share bisection bandwidth	Per-port WFQ
	SecondNet [36]	Fixed	At host	No	Yes	Tenant with traffic matrix	Single-path routing
	VLAN Isolation	Fixed	No	No	Yes	Network per tenant	Per-port WFQ
Entity-based	Seawall [84]	Proportional	Yes	Yes	No	Per-VM charging & shares	Large tenant bias
	FairCloud [72]	Proportional	Yes	No	No	Per-VM charging & shares	Tree-based topologies

At one end of the spectrum, strict systems like Oktopus [8] and Proteus [101] perform admission control, causing them to reject new tenants when insufficient resources are available to guarantee performance. Regardless of the physical reality, both systems export to tenants an abstraction of a single non-blocking switch to which all of the tenant VMs are connected at a guaranteed rate. Oktopus also permits tenant requests for multiple such virtual switches, organized into clusters, which is beneficial for tenants whose services exhibit non-uniform communication locality. Oktopus connects a tenant's clusters via an additional top-level virtual switch whose rate is reduced by an "oversubscription factor." Proteus extends Oktopus's model to account for the variance in tenant bandwidth requirements over time, filling the gaps by allowing the network to service more tenants simultaneously.

On the other end, proportional allocations avoid the need for admission control and aim to provide fairness between tenants. Such systems generally achieve high overall network utilization by giving up predictable performance in favor of work conservation. In this category, Seawall [84] associates an administrator-defined weight value with each network entity (typically VMs). A novel feedback loop enforces weight-proportional bandwidth allocations across paths that carry traffic for multiple entities. Another system, Netshare [49], uses a centralized bandwidth allocator to implement weighted hierarchical max-min link sharing between tenants. Max-min sharing ensures that every tenant will receive a fair portion of link capacity while maintaining high link utilizations.

Other efforts occupy middle ground in the design space. Gatekeeper [76] exports an Oktopus-like non-blocking switch abstraction and enforces tenant minimum rates with admission control but enables weighted fair sharing for tenants to exceed their guaranteed rates when excess capacity is available. Faircloud [72] formally characterizes the network isolation design space in the context of five desirable sharing properties:

work conservation, strategy-proofness, utilization incentives, communication-pattern independence, and symmetry. They describe three isolation models and explicitly analyze their implications for predictable performance and high utilization.

Finally, VINI [10] represents a vision of a virtualized network whose goal is to provision a shared environment in which researchers can perform network experiments that are both realistic and repeatable. To support experimentation, VINI targets execution on PlanetLab [71], an experimental wide-area research testbed, rather than data center network environments. Others [75, 104] take an analytical approach, supported by simulations, to design virtual network embedding algorithms that map virtual networks onto a shared physical substrate. SecondNET [36] applies virtual network embedding to the data center with the intent of supplying applications with their exact capacity needs. It uses a novel MPLS-based source routing scheme, but requires users to have detailed information about application behavior to take full advantage of the benefits.

Generally, the systems described in this section enforce their isolation models through the collective behavior of traffic policers across end hosts or, more rarely, within network devices [49, 72]. While some systems set policer limits statically at the time of allocation, many utilize mechanisms like work conservation [49, 76, 84] or distributed rate limiting [8, 76] that must observe and react to dynamic traffic characteristics to enforce their performance guarantees.

2.4 Network Programmability

The traditional model of using vendor-supplied software to manage closed network hardware severely hampers the ability of providers to construct and deploy innovative network solutions. As with commodity end host platforms, open, programmable network devices equip data center providers with the customizability they need to develop unconventional network architectures. Despite software-defined networking's current popularity, proposals for innovative network control frameworks predate SDN developments. This section summarizes the evolution of network programmability and describe the features available to current platforms.

An early model, Active Networking [99] provides differentiated network behavior to applications in response to custom, user-supplied forwarding directives. Several projects fall under the active networking title, the most prominent of which is ANTS, a Java-based system that propagates per-application forwarding instructions in augmented packets called capsules. While both ANTS and our architecture aim to empower applications with improved control, we assume a different role and level of involvement for service operators. ANTS requires that applications choose their forwarding schemes and appropriately construct capsules, whereas in our architecture, tenants make only simple, high-level requests for resources.

Contemporary to ANTS, Tempest [77] allows users (or network operators) to create virtual private networks over an ATM substrate. Tempest partitions switches into “switchlets”, which appear to users as fully-functional switches under their private control. Similar to modern OpenFlow switches, a designated component on each switch translates high-level instructions into the appropriate switch-specific configuration. Tempest does not prescribe which party is responsible for assigning switchlet configurations, opting instead to define a general framework that can be flexibly controlled by either users or network operators, depending on the desired use case.

The authors of FIRE [69] observe that as routing algorithms have evolved, network providers have become interested in specializing routing behavior for multiple classes of traffic. FIRE decouples routing protocol state propagation from the algorithms that use link metrics to compute forwarding tables. It allows multiple routing algorithms to execute simultaneously, with a filter for classifying traffic to the appropriate algorithm instance. FIRE’s architecture is modular, permitting providers to load at new routing algorithms and classifiers at runtime.

More recently, Tesseract [102] proposed the adoption of a “4D” control architecture [33] to separate the logically-distinct decision, dissemination, discovery, and data planes of a network. The SANE [54] and Ethane [15] projects adopted 4D’s principles and made centralized control practical in the context of enterprise network security. SANE aims to prevent network attacks and mitigate the threat of compromised devices, while Ethane enforces network connectivity policies. Both focus on user authentication and employ a domain-specific policy language to express permitted/prohibited connectivity.

These 4D systems laid the foundation for future centralized software-defined networking platforms. Today, with hardware support from multiple vendors [79], the availability of several open source controller software projects [24, 34, 73], and adoption in production networks [44], OpenFlow [55] is rapidly gaining traction as the de facto SDN protocol.

2.4.1 Higher-level Programming Constructs

While SDN and OpenFlow deliver programmability for network hardware, the API is sparse – akin to a RISC assembly language for the network. This interaction may suffice for data center providers, but for service operators, an attractive aspect of centralized SDN is the potential to build network control software out of high-level, modular components. Here, we describe abstractions and programming languages that simplify SDN’s details.

Built atop OpenFlow, FlowVisor [83] provides strict application partitioning by dividing the network into “slices.” Each slice maintains private management routines and is isolated from other slices via independent hardware queues in the switches. This isolation enables multiple applications to share the physical network while unrelated

OpenFlow controller software manages each slice. In contrast, Onix [47] presents an all-purpose controller framework designed to be shared by simultaneously executing applications. Like our architecture, Onix provides a virtualized graph-based view of the physical network for applications to manipulate. However, Onix requires that applications mediate their own resource usage, while our architecture automatically enforces performance isolation between applications that share the network.

Other recent projects address the need for higher-level SDN programming languages. These languages aim to reduce the complexity of constructing and maintaining SDN controller software. Nettle [94] is a domain-specific, functional language for programming OpenFlow devices. Embedded in Haskell, Nettle allows network operators to write declarative programs that receive network events and respond by issuing OpenFlow commands. The Frenetic [28] project enables parallel composition between reusable modules, a declarative query language for classifying and aggregating network traffic. It uses a novel runtime system to manage parallel module execution, but it does not by itself provide performance isolation across modules.

Pyretic [59] extends the ideas of Frenetic with an imperative Python-like language for sequential module composition. Pyretic also allows for the attachment of virtual traffic header fields, enabling network control software developers to extend packet processing with custom, high-level metadata. Together, these efforts enable the composition of applications from the parallel and sequential execution of modular components, but efficiently compiling their directives into forwarding hardware remains challenging.

2.4.2 General Purpose Network Processing

Most SDN devices have severely constrained resources for control plane processing. Furthermore, OpenFlow hardware exposes limited visibility into the device with a narrow interface consisting of data plane rules and output actions. Several research efforts have developed alternative platforms that enhance network devices with high-performance, general purpose processors. They aim to construct a fully programmable platform with low-level hardware interaction just as others have have invested significant effort towards constructing high-speed routers from general purpose hardware [23, 40, 89].

SideCar [85] and ServerSwitch [52] propose architectures that augment traditional networking hardware with commodity CPUs to improve network programmability. Each system approaches the problem from a different angle – SideCar adds high-performance general purpose hardware to network devices, while ServerSwitch incorporates specialized switching silicon into traditional server architectures. NaaS [19] advocates exposing already present in-network processing elements to tenant applications, allowing them to execute custom parallel packet processing code.

2.4.3 Middlebox Services

Data center providers deploy middleboxes to supplement the network with additional, often location-sensitive functionality such as firewalling, intrusion detection, and load balancing. APLOMB [82] surveys common enterprise middlebox deployments and proposes outsourcing their functionality to cloud infrastructure. They introduce novel mechanisms for efficiently redirecting traffic between the cloud, the enterprise, and the general Internet. CoMB [80] makes the observation that many middleboxes require common functionality (e.g., packet classification, enforcement, etc.) that could be made modular. It aims to consolidate middleboxes onto a shared set of hardware resources that are assigned via constraint satisfaction at a central controller.

In the context of network security, Fresco [86] provides a framework for composing security modules and takes an authority/priority-based approach to resolving conflicts at the level of individual rules, rather than modules. In addition to basic forwarding, Fresco can statelessly redirect packets, mirror packets to an external packet analysis system, or quarantine packets to a restricted subset of the network.

CloudNaaS [11] provides a flexible language for tenants to request middlebox interposition and custom endpoint addressing in virtual networks. It leverages SDN by compiling tenant requests into a fault tolerant set of forwarding rules, but it does not take advantage of dynamic network traffic state or in-network packet processing.

2.4.4 Other Objectives

Finally, other systems coordinate network resource usage in support of other high-level objectives. DevoFlow [21] reduces the burden on the centralized SDN controller by delegating functionality to network switches. However, DevoFlow focuses on improving the performance of control tasks such as setting up hardware flow table entries or collecting statistics rather than executing higher-level application logic. Towards a similar goal, DIFANE [103] treats the global set of switch rule memory like a large distributed cache. It cleverly constructs wildcard rules to keep all traffic in the data plane, yielding a better "division of labor" between controller and switches and freeing the controller to handle other concerns.

Acknowledgements

Chapter 2 contains material that has been submitted for publication in the Proceedings of the ACM Symposium on Cloud Computing (SOCC), 2013. "Blender: Mixing Data Center Network Isolation Models". Webb, Kevin; Yocum, Ken; Snoeren, Alex. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Task Switching for Data Center Networks

This chapter lays the groundwork for *task switching* (TS), a fundamentally different way for data center tenants and their applications to interact with the network. The task switching abstractions serve as the foundation for our Blender and Omakase models. We first present task switching and its design goals, followed by an explanation of the task-switched network architecture. Finally, we describe task compilation and the life cycle of a task on the network.

3.1 Motivation and Goals

Application performance and network management suffer from a one-size-fits-all design prevalent in today's data center networks. As outlined in Chapter 2, researchers have proposed many systems that offer advantages to data center networks. While they generally introduce performance improvements, each proposal affects tenants differently. For any particular system, some tenant services may benefit strongly, while others might perform as well without it, making the investment unnecessary for that tenant.

As a departure from traditional network architectures, a task-switched network supports multiple, simultaneous application-specific tasks. Within each task, the executing application can define distinct topologies and routing conventions specifically tailored to its unique resource requirements, providing enhanced performance at the expense of certain attributes that may not be critical to the task at hand. Thus, the data center network no longer needs to balance a single solution for routing consistency, failure resilience, high performance forwarding, flexible policy enforcement, and security across all the applications in the data center.

Task switching addresses the challenge of providing network virtualization in multi-tenant data center network environments. While end host virtualization allows any server to host any VM, the network is often a key performance bottleneck [39]. The majority of proposed data center network architectures try to decouple placement from performance by maximizing bandwidth between server pairs through symmetric topologies. While this strategy admits considerable flexibility in assigning work across the data center, there are cases in which skewed communication patterns dominate. We argue that for these workloads it is far more effective to select application-specific resource allocation systems to make efficient use of the network's physical resources.

Individual tenants host a variety of services that each present different networking demands. For example, the hosted services themselves may be clustered, multi-tiered, or replicated. Each of these design patterns performs best with a distinct set of resources—a low-diameter mesh, a high-bandwidth tree of arbitrary depth, and an edge-disjoint, redundant multi-graph, respectively.

Additionally, real-world data center infrastructures grow and evolve, often becoming less homogeneous and symmetric. This heterogeneity may result from growth, limited budgets, physical wiring constraints, or the presence of dynamic link allocation via optics [26, 97] or wireless [38]. As the topology distorts, routing designs based upon a systematic network design may begin to perform poorly—or not at all. Task switching functions over any physical topology, allowing applications to optimize for the network at hand. Finally, the needs of data center applications are changing quickly. For example, Amazon recently introduced explicit support for high-performance computing (HPC, applications that often have latency-bound phases) instances on EC2, presumably in response to sufficiently strong economic incentives.

For task switching, we assume a data center environment that is controlled by a single administrative entity, which we call the *network provider*. Whether the data center supports private services (e.g., Google or Facebook) or public cloud computing (e.g., EC2 [5] or Azure [57]), we assume that multiple applications and services wish to share the physical network infrastructure. We call the users who operate these applications *service operators* and assume they are distinct from those running the network (even if they are working for the same company in the case of a private data center). In this setting, we aim to achieve the following goals with the task switching network architecture:

- **Modularity for network providers:**

  A key goal of network virtualization is to set the application free from the details of the physical network, giving applications the freedom to connect any set of servers as if they had a dedicated network [39]. Task switching allows network providers to realize this vision for data center networks by decoupling tasks' virtual networks from the physical reality. Task switching should not mandate any particular physical topology, instead it should *map* virtual networks onto any given physical topology to meet both per-application and network-wide performance objectives.

  Furthermore, be it to support performance isolation (Blender) or general functionality (Omakase), task switching's resource allocation policies should be modular and easily extensible. Providers should have the flexibility to compose powerful network primitives (e.g., rate limiting, flow statistics, etc.) in support of simultaneously executing tasks with autonomous resource allocation objectives (Section 2.3.1).

- **Simplicity and flexibility for service operators:**

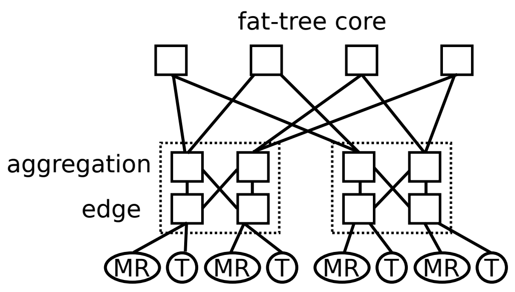
  Service requests should be simple and concise, allowing service operators to specify their high-level task objectives (i.e., what they need from the network), not the details of how to get it. They should *not* need to supply (potentially secret) knowledge of the underlying physical network or an excessive amount of detail regarding their task's traffic characteristics (i.e., a traffic matrix) to effectively tailor their task's virtual network to their needs.

- **Practical and scalable:**

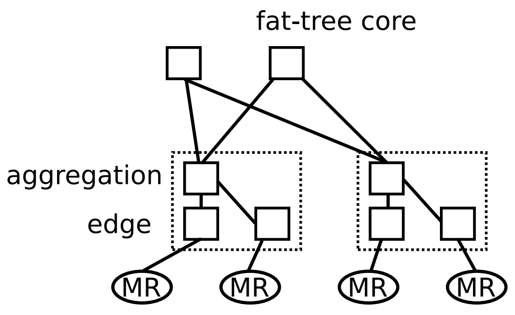
  The task switching architecture should account for resource limitations in network device storage and processing. Task switching should execute on realistic hardware that is available now or in the near future, enabling real implementations at data center scales.

3.2 Task Switching Architecture

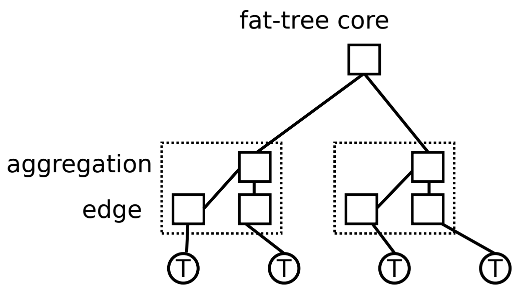
A task-switched network allows applications to create multiple, custom tasks to meet their networking requirements throughout their lifetime. Unlike other SDN architectures that allocate at the granularity of flows [3, 15], the fundamental unit of allocation is the network task: an autonomous subset of the network's resources resembling the virtual network slices described in Section 2.4.1.



(a) The physical data center network.



(b) A task for a MapReduce application.



(c) A task for a trading platform.

Figure 3.1. Optimizing tasks for individual applications, task switching finds a fat-tree subgraph for MapReduce (MR) and an isolated spanning tree for the trading platform (T).

Tasks specify the set of communicating end hosts within the data center, a desired virtual topology to construct between those hosts, and a resource allocation system to manage link resource allocation and route selection. A logically centralized (but possibly replicated) task server registers each allocation system, compiles tasks, and manages individual task deployment. The task server maintains all of the state for a task, which includes the physical network representation, allocated virtual networks, and the state of the network devices.

As an example, consider the fat-tree data center network in Figure 3.1(a) connecting eight physical hosts. Various emerging multi-path resource allocation systems support well-provisioned networks [3, 42, 64, 91], but their unified allocation systems remain blind to individual application needs. Such networks may host a range of applications inside distinct VM pools, such as a bandwidth-hungry MapReduce/Hadoop cluster (MR) and a queuing-sensitive trading platform (T). While the MapReduce application is generally bottlenecked by the available bisection bandwidth in the network, trading platforms demand the consistent low-latency performance of isolated network paths [7].

In contrast, a task-switched network treats these two applications as distinct tasks. Each task runs an instance of a particular allocation system that best addresses the communication pattern and preferences of that particular task. An allocation system includes an *allocator* that determines the subset of the physical network that will connect the application endpoints in its task. In this case, the MapReduce task in Figure 3.1(b) searches for high-bandwidth physical paths between mappers and reducers to optimize the performance of the shuffle phase. In contrast, the trading platform in Figure 3.1(c) allocates for isolation, building an exclusively owned spanning tree. Allocation systems also define a set of route selection rules that allow switches to make application-specific forwarding decisions across multiple paths.

Allocators take as input the node set, a desired virtual topology, and a view of physical network connectivity from the task server. The allocator then maps one or more physical paths to each link in the virtual topology to achieve its performance objectives. Allocation occurs in an on-line fashion, processing task requests in the order they arrive.

Figure 3.2 sketches an overview of the task switching architecture. The task server guides allocation in many ways. First, the server may prune links or switches from the physical network view before passing it to the allocator, allowing network administrators to export different network views to each task. This mechanism makes it trivial to physically separate traffic between tasks. For example, the task server can remove the spanning tree links in Figure 3.1(c) from the network view passed to the MapReduce allocator, rendering it impossible for MapReduce tasks to consider executing on that subsection of the network. The task server may also perform admission

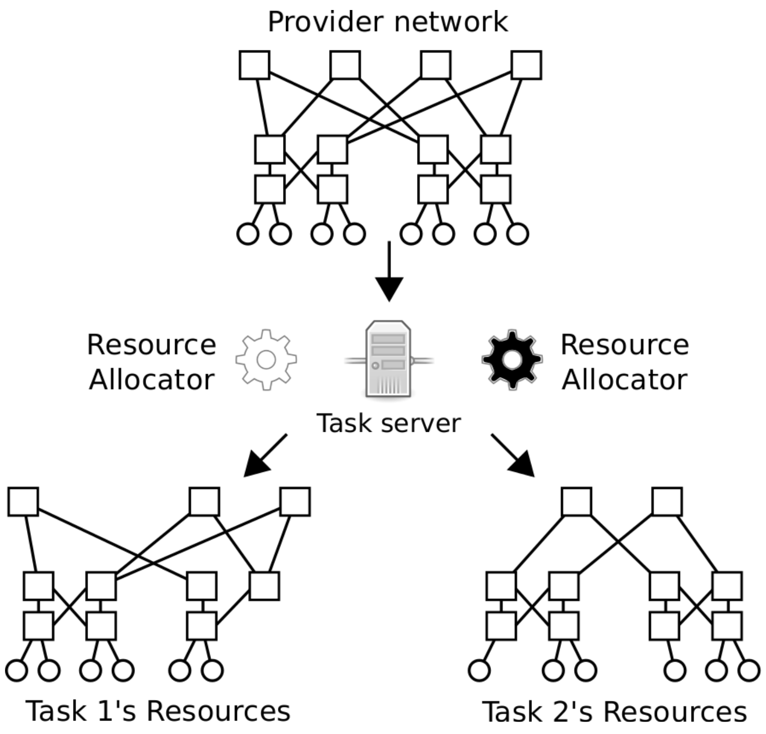


Figure 3.2. A task server mediates access to the network, compiling individual tasks and performing admission control.

control on the proposed virtual topologies, and that admission control could consider a wide range of metrics: It may limit tasks based on current network load, task type and count, or expected monetary return of the task. The task server might also choose to revoke tasks.

The increased flexibility of task switching may also make other administrative tasks easier. For instance, tasks and resource allocation systems represent two new units by which the network may be controlled. This architecture permits a number of mechanisms that act on these units of control and enables a rich policy space. For example, the network provider might migrate tasks for upgrading or expanding the current network. This modularity also increases network operation transparency, allowing routing elements to attribute activity to one task or multiple tasks. Attribution gives administrators an increased ability to diagnose errors, and makes it easier to assign blame (or

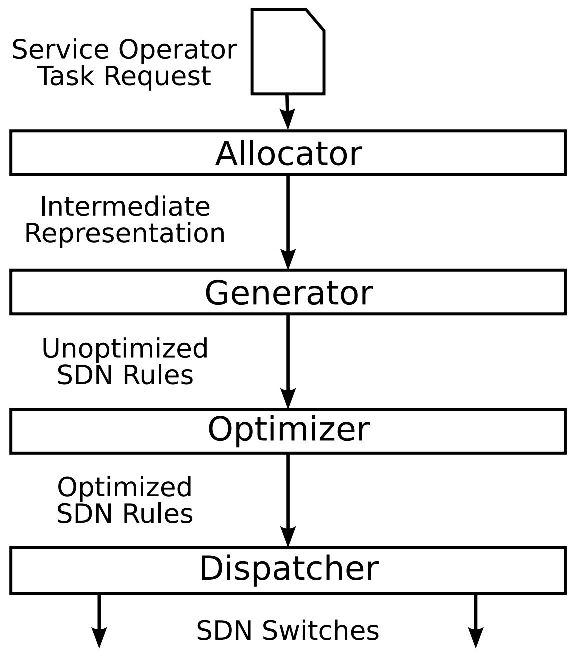


Figure 3.3. The life cycle of a task is analogous to program compilation, with multiple stages refining a high-level input until it can be executed in hardware.

praise) to the end user who registered the offending task. Further, such information is critical for allowing administrators to determine policies for task admission and network capacity planning.

3.3 Task Compilation and Life Cycle

The task switching task server must compile, install, and monitor multiple application tasks. This design builds on a software-defined network (SDN) infrastructure whose switches accept rules that dictate how packets move through the network [15, 55]. Figure 3.3 depicts the sequence of steps involved in processing an application's task request, which is analogous to traditional software compilation. Each processing step refines the request, facilitating its transformation from an abstract set of goals into a collection of low-level forwarding rules that switching hardware can interpret and install to instantiate the resulting task.

A task begins as a service operator's high-level description of his application's desired network properties. This description, which we call a task request, is analogous to the source code presented to a traditional compiler. The exact content of the request will depend on the service operators goals, but it would likely contain the desired allocator, topology, routing characteristics, performance requirements, and other high-level objectives for the task. Requests may optionally include detailed performance characteristics (e.g., the expected traffic matrix), but this level of detail is neither required nor expected to be available from most service operators.

The compilation process starts when a service operator submits the request to the task server, where it is received by the specified allocator. The allocation phase operates over the task server's abstract view of the physical network in an attempt to satisfy the client's task request. As it proceeds, the allocator builds a virtual network topology and annotates the physical view's links with the results of the allocation, similar to producing an intermediate representation. Annotations include adding the request's unique identifier to each of the links, updating the number of allocated routes over the link, and recording allocated link properties. The annotations inform future allocations of the allocator's decisions, allowing them to avoid resource conflicts.

Next, the task server delivers the annotated virtual network to a rule generator, which translates the newly allocated view into a set of SDN hardware rules. The generation process is analogous to a compiler converting an intermediate representation into low-level machine code. The resulting rules consist of a set of identifying packet headers to match, an output action to perform for matching packets, and other directives to ensure that performance objectives are met (e.g., rate limiting or QoS).

SDN switches have limited capacities in their forwarding tables. Therefore, following generation, the task server executes rule optimization routines. The optimization phase conserves space in these switch tables by intelligently marking rules for the correct forwarding table and combining rules into a single wild-carded entry, when appropriate. Such optimizations improve the scalability of the system, allowing more rules, and thus more tasks, to be installed. Finally, after being generated and optimized, the task's rule set is given to a rule dispatcher, which methodically pushes them into the switches' forwarding tables. The dispatcher manages rule installation consistency, accounting for the hardware's hardware installation rate.

3.4 Summary

A variety of technologies now exist to allocate resources to tenants across the multiple paths found in well-connected topologies. While they increase performance by spreading flows for all applications in a similar fashion, such unified treatment makes it difficult or impossible to deliver qualitatively different paths to applications. Task switching rejects this one-size-fits-all approach to path selection and provides applications with the ability to have a stake in their resource allocation decisions.

The next chapter describes a feasibility analysis of the task switching architecture using three example resource allocation systems. In subsequent chapters, we perform a simulated analysis of task switching (Chapter 4), introduce a core model for managing task allocations over a graph network representation in support of tenant performance isolation (Chapter 5), and extend the model for composing general purpose network functionality (Chapter 6).

Acknowledgements

Chapter 3 contains material as it appears in the Proceedings of the USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE) 2011. "Topology Switching for Data Center Networks". Webb,

Kevin; Snoeren, Alex; Yocum, Ken. The dissertation author was the primary investigator and author of this paper.

Chapter 3 contains material that has been submitted for publication in the Proceedings of the ACM Symposium on Cloud Computing (SOCC), 2013. “Blender: Mixing Data Center Network Isolation Models”. Webb, Kevin; Yocum, Ken; Snoeren, Alex. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Task Switching Simulation

This chapter describes a simulation framework to explore the viability of the task switching design. We examine three allocation strategies, their objectives, and the metrics used to evaluate them. Our results show that task switching allows allocators to effectively optimize for different metrics, even for a randomized placement of hosts across the physical infrastructure.

4.1 Allocation Strategies

Task switching allows network operators to customize tasks along three primary axes: virtual topology, choice of allocator, and route selection rules. Here, we set aside route selection and employ simple, hash-based multi-path route selection for all tasks. Additionally, we only consider logical mesh networks in which every pair of end hosts can communicate with one another. Other virtual topologies, such as rings (for chain replication) or trees (for aggregation or file distribution), could leverage allocation to customize the arrangement of the topology, e.g., to build an efficient aggregation tree. Within these constraints, we present three example allocators to illustrate how task switching might optimize a tenant's network for resilience, isolation, or bandwidth. We return to to examine more pragmatic allocators along with a model for customizing route selection and defining new virtual topologies in Chapter 5.

Allocators use three components for assigning resources to tenants. First, each allocator requires one or more metrics to define an objective function that drives the resource assignment process. The second component is an allocation algorithm to maximize or minimize the objective function. The third component annotates and optionally filters the network for future tasks. Since allocation is an on-line process, an allocation's goodness may decrease as other allocations (optimizing for different metrics) are made. Thus the allocator may wish to store additional annotations on the physical network to inform future allocations. The topology server may optionally filter links based on these annotations, ensuring that other allocators do not decrease the quality of prior allocations (for an example see Section 4.1.3).

Formally, we represent the physical network as a graph $P = \{V, E\}$, where V is the set of hosts and switches and E is the set of physical links. A task's virtual network $N = \{H, L\}$ is a set of hosts $H \subseteq V$ and virtual paths L connecting the hosts in H. Given a view of the physical network, P^{view}, an allocator chooses which links to use for each of a task's inter-host paths. The topology server maintains a set of annotations for each physical link, including the total number of tasks using the link (T_i), its physical capacity (C_i), and a number of *claims* (M_i) made to that capacity (i.e., the number of task paths that traverse it).

4.1.1 Bandwidth

Maximizing capacity is often a core objective of traditional network architectures. These topologies are well suited for parallel data processing tasks that require high bisection bandwidth in support of low all-to-all transfer times. Other bandwidth-oriented tasks include data backup, video and audio serving, and VM image distribution.

Metric: We use bisection bandwidth as the metric to evaluate the performance of the bandwidth allocator. The bisection bandwidth of a topology is the amount of bandwidth a bijection of hosts can transfer to one another; it is a rough measure of a topology's ability to handle many concurrent transfers.

Allocation: An allocation strategy may consider either single or multiple path solutions, where multiple physical paths support the virtual path. In either case, one allocation strategy is to maximize the total flow possible along links (end-to-end paths) in the virtual topology. However, even a single-path solution must consider how other virtual paths in this task have been mapped, otherwise many virtual paths could be mapped onto the same physical links. This approach can be modeled as a maximum multi-commodity flow problem, where polynomial time solutions exist when allowing fractional flow allocations.

However, for simplicity we approximate this allocation by using a single-path allocator. This allocator uses a maximum spanning tree to find the current maximal path [53]. Such an approximation is reasonable, as many recent performance isolation models operate over singly-rooted trees [8, 72].

Physical network annotation/filtering: The bandwidth allocator depends on the number of claims on each link to determine its available bandwidth. The topology server manages link claim annotations, M_i, which the allocator uses to set the available capacity on links proportionally as $\frac{C_i}{M_i}$. Bandwidth tasks increment the M_i annotation by one for each physical link mapped to a virtual path. In contrast, the subsequently described resilience and isolation allocators increment M_i by $1/N$ when a virtual path is split over N physical links, dividing bandwidth equally across them. Annotating in this fashion allows the bandwidth allocator to take advantage of the capacity left by allocations that are not bandwidth constrained. This strategy assumes that the network has the ability to rate-limit end hosts, perhaps using emerging VM-based technologies [36, 84].

4.1.2 *r* resilience

A tenant may also wish to increase the overall physical path diversity available to its virtual topology. For example, consider aggregation trees used for scalable monitoring [51]. These tasks are willing to traverse longer paths in return for increased failure resilience, ensuring more hosts are connected during failures or congestion. The large numbers of components in modern data center networks means that some level of failure is virtually always present [32]. Failures are perhaps even more of a concern with emerging topologies that depend on end hosts to participate in switching [35, 37].

Metric: Here we measure resilience as the number of cuts *r* in the physical network required to break a virtual path. Since hosts have a single up-link to their top-of-rack switch in the physical topologies we study, we ignore cuts to those access links for pairs of hosts on different switches. We note that this approach provides an aggressive notion of resilience, providing *r* disjoint paths (not including access links) between end hosts.

Allocation: Here we use shortest paths to find a suitable set of *r* paths between end hosts. For each virtual path we repeatedly find a shortest path with respect to hop count, add it to the set of possible paths, and remove its links from consideration for future paths between the same pair of end hosts. The task parameters may also specify an average, rather than minimum, resilience that allocation must reach. Specifying an average allows some virtual paths to be backed by fewer than *r* disjoint physical paths, increasing the likelihood that an allocation will find enough paths to succeed.

Physical network annotation/filtering: Beyond setting the link claims as described above, this allocator does not require additional annotations. Future allocations will not decrease the resilience of allocated tasks.

4.1.3 *k* isolation

Isolation between tasks may be used to ensure consistent levels of network performance, or to isolate critical services from potentially disruptive external traffic sources. However, modern data center switches are often limited in their ability to provide per-flow isolation. While there may be hundreds to thousands of separate tasks, current switches support only a handful of fair queuing classes [84]. Tenants may use the *k*-isolation allocator to ensure sufficient resources exist to limit the degree of sharing with other tasks.

Metric: We measure isolation on each physical link as the number of tasks with a virtual path on that link. Here each task specifies the maximum number of other tasks that may share a link, k. k may either reflect the maximum number of service classes the physical network can support or it may be set to one to provide complete isolation from other traffic. With this metric, k limits sharing on a per-link basis. A more restrictive scheme could provide isolation by limiting the number of tasks that can share a switch. This restriction would ensure further task separation and eliminate any potential for inter-task interference in switches at the cost of more quickly exhausting network resources.

Allocation: To increase isolation, we wish to find a topology that connects the task's nodes that is minimally shared with other tasks. We approximate the aggregate level of sharing by summing the total number of tasks allocated across the chosen physical links. This task is equivalent to finding a minimum-cost tree connecting the hosts, a Steiner tree on an existing graph, an NP-hard problem. We employ the minimum spanning tree heuristic, with a worst-case performance ratio of two [107].

The allocator computes the minimum spanning tree on the physical network, using the number of current tasks as the edge weight. The allocator then removes all physical links and switches from this spanning tree that are not used in the paths between nodes in the task. Each link used in the mapped topology increases its task count. In the end, the allocator produces a task whose virtual paths all map to physical links whose task count (T_i) is less than or equal to the allocator's sharing parameter k.

Physical network annotation/filtering: To prevent subsequent allocations from violation the isolation requirement, the topology server remove links from consideration. In this case, the topology server first removes links whose number of tasks is already k. To do so, the topology server records the minimum k used by an isolation task for each link. $k = \infty$ for links with no isolation tasks. Note that removing links may disconnect P^{view} and subsequent allocations may fail.

4.2 Simulations

A fundamental question posed by task switching is whether allocating paths for specific objectives can improve the networks for all tasks relative to a unified resource allocation system. To begin to answer, we build a simulator that takes as input a physical network graph, P, and multiple tasks. We simulate the three allocators described in Section 4.1 and report how well each task meets its objective metrics (and those of the other allocators).

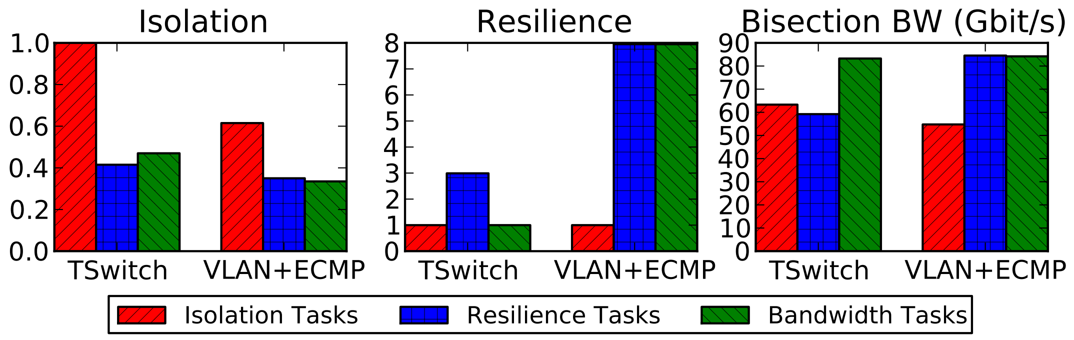
We study task switching on a $k = 16$ fat tree with 16-port switches and 16 pods. All links are 1 Gb/s in capacity. We create a pool of tasks, each allocated using one of our three allocators. Each task uses an exclusive set of end hosts randomly distributed across the physical network. We use this "network oblivious" host-to-task assignment to determine if task switching can meet different objectives without optimizing physical layout. All tasks have equal node count and use a mesh virtual network. Finally, our experiments allocate isolation before resilience and bandwidth tasks, ensuring that those tasks find $k = 1$ isolations.[1]

Clearly the results depend on the particular physical topology and set of tasks. Hence, the resulting metrics are only meaningful in comparison to an alternative design. While traditional Ethernet's single spanning tree is an obvious choice, it is an exceptionally weak strawman as it fails to take advantage of the path diversity and resulting bisection bandwidth made available by recently proposed physical topologies.

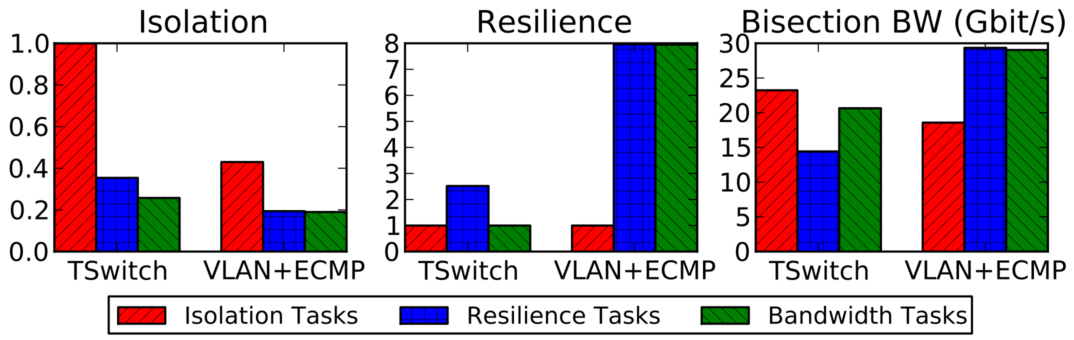
Instead, we compare our task-switched allocations to a unified resource allocation system modeled after recent proposals such as Trill [91], 802.1aq [64], and Cisco's FabricPath product [18]. These systems route layer-2 frames using multiple shortest paths calculated by instances of the IS-IS routing protocol. We emulate this ECMP approach by calculating all shortest paths in physical network P for each pair of end hosts. We then assume that the pair can achieve the maximum flow along the union of those paths. With this assumption, performance outperforms standard hash-based ECMP, which is oblivious to available capacity and limits the number of considered paths (16 for FabricPath). We simulate a VLAN for each task, using a portion of the k-isolation allocator's logic to place the task as if an administrator had planned the task. However, unlike the full k-isolation allocator, ECMP emulation does not prevent successive tasks from sharing links with other VLANs.

We evaluate allocations based on metrics of our allocators. We calculate isolation using the number of virtual paths each task assigned to each physical link. For a given task, we calculate isolation as the ratio of the task's paths to the total path count on the link, averaged across all physical links. A value of one indicates a highly isolated allocation while near zero values indicate that other tasks dominate the link. We measure resilience as described in Section 4.1.2.

[1]This ordering emulates a topology server that runs isolation tasks on a network view that only shows isolation allocations. It would then re-allocate other tasks that conflict with the new mapping.



(a) Performance of a 2i-2r-2b task mix.



(b) Performance of a 7i-5r-4b task mix.

Figure 4.1. Task switching versus ECMP emulation on a $k = 16$ fat tree with 1024 hosts.

To measure the bandwidth of the allocated virtual network, we consider each task's *effective* bisection bandwidth. Since task allocation often results in asymmetric topologies, we calculate bisection bandwidth as the average maximum flow between two randomly chosen sets of $|H|/2$ nodes (repeated 200 times). Note that maximum flow depends upon the capacity of the physical links in the allocated virtual network. To take into account the existence of the other tasks, we weight the physical capacity by the total claims (Section 4.1.1) made on this link by this task divided by all claims made on the link.

We first investigate allocations for six tasks: two isolation, two resilience, and two bandwidth (2i-2r-2b). This mixture represents a balance between the three task classes with a low overall task count. Figure 4.1(a) shows the average results for each metric for the isolation, resilience, and bandwidth tasks respectively. Looking at task switching ("TSwitch"), all tasks achieve their goals. First, isolation tasks achieve their target ($k = 1$). In contrast, "VLAN+ECMP" provides less isolation for all tasks, though our allocator increases isolation relative to the other tasks. Interestingly, task switching provides increased isolation for the resilience and bandwidth tasks as well. This effect is likely because we remove isolation task links from consideration and our bandwidth allocator uses fewer paths than ECMP.

Second, in task switching, resilience tasks receive their target of exactly $r = 3$ disjoint paths through the topology. ECMP on the other hand treats resilience and bandwidth tasks identically, spreading virtual paths across as many shortest paths as exist, giving both high resilience. Finally, the task-switched bandwidth tasks receive the highest effective bisection bandwidth, illustrating that the other allocators are giving up bandwidth relative to their optimization goals.

It is also important to qualify the kind of bandwidth ECMP emulation reports. In a fat tree, this emulation assumes near optimal effective bisection bandwidth by using every minimum hop-count path, and the number of multiple paths for ECMP emulation is unrealistically large (an average of 60 paths with a standard deviation of 13.5) per virtual path for both 2i-2r-2b and 7i-5r-4b task mixes. Additionally, the task-switched isolation tasks receive more bandwidth than their counterpart tasks under ECMP emulation. More importantly, that bandwidth *is not shared* with any other tasks mapped to the topology.

Figure 4.1(b) shows the results of allocating seven isolation, five resilience, and four bandwidth tasks (7i-5r-4b). This mixture represents an imbalanced task distribution in which a significant fraction of the network is allocated exclusively to isolation tasks. By using seven isolation tasks, seven of the eight uplinks of the fat tree's edge switch cannot be allocated to other tasks when using task switching. This configuration reduces the ability of task switching to deliver high effective bisection bandwidth to the bandwidth task, a 31% decrease relative to the ECMP emulation.[2] Despite the additional constraints, the task-switched isolation tasks continue to receive superior capacity; a trade-off a unified resource allocation system cannot achieve. The benefit could be further improved with a multi-commodity flow bandwidth allocator, rather than our current approximation scheme.

Finally, task switching and ECMP utilize the network similarly for the 2i-2r-2b mix. However, task switching the 7i-5r-4b task mix allows 10% of the links to go unused in the fat tree. This reduction in utilization is essentially the penalty of supporting many isolation requests.

4.3 Summary

Task switching formalizes the simultaneous use of multiple resource allocation mechanisms in a data center network, allowing network providers to define multiple resource allocators and tenants to request individualized tasks. This chapter used simulation to illustrate how task switching can provide both network efficiency and individual application performance to admit flexible network management strategies. We described an abstract set of allocators that allow applications to take advantage of the paths that others choose not to utilize. Further, our simulated results show that we can leverage almost the full diversity of a fat tree topology to meet differing application objectives.

The next chapter introduces Blender, a programming model with a concrete prototype that embraces the task switching framework. Blender moves beyond abstract allocators and assigns resources according to the objectives of practical performance isolation models proposed in recent literature. It defines a straightforward interface for providers to author allocators and manipulate the graph representation of the network.

[2]Note that by hosting 16 tasks of 64 nodes, the maximum possible bisection bandwidth is 32 Gb/s.

We evaluate a Blender prototype system that handles crucial implementation challenges, including managing the costs and complexity of dynamic reconfiguration in real switching hardware.

Acknowledgements

Chapter 4 contains material as it appears in the Proceedings of the USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services (Hot-ICE) 2011. “Topology Switching for Data Center Networks”. Webb, Kevin; Snoeren, Alex; Yocum, Ken. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Blender: Mixing Data Center Network Isolation Models

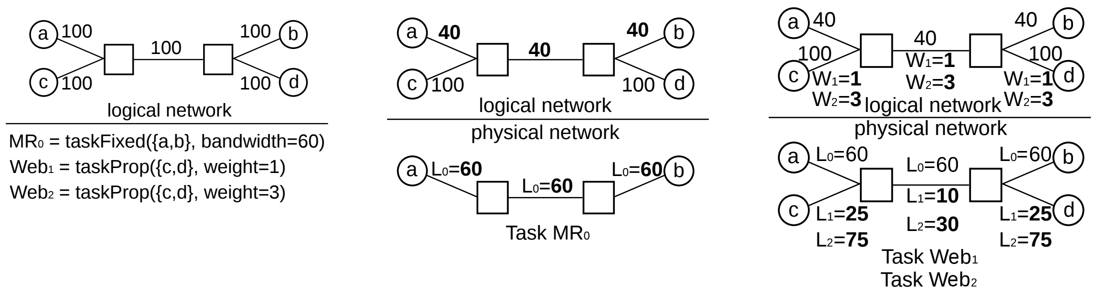
Large data centers host thousands of services and end-user applications. For efficiency, they multiplex tenants across the physical infrastructure, leveraging server virtualization to carve out isolated units of CPU, memory, and storage. In contrast, tenants typically receive only loose, qualitative descriptions of network performance. This imprecision leaves data center providers at the whim of their tenants and forces tenants to design services for worst-case network scenarios. With unpredictable network loads from a diverse group of tenants, today's data center providers must accept hot spots (i.e., tenants pay to wait [96]) or monitor and dynamically adjust VM placement [56], application components [105], or network flows [3] to improve utilization.

Performance isolation in multi-tenant neworks has received considerable attention, and many network allocation schemes have been proposed that provide different degrees of isolation [8, 36, 49, 72, 76, 84]. For example, some schemes provide predictability via fixed performance guarantees [8, 36], but limit the number of concurrent tenants placed on the network to ensure that guarantees are met. Other schemes provide proportional network shares, allowing tenants to receive performance relative to use [49, 72, 76, 84]. In fact, these are fundamental trade-offs that any isolation model must make—no single network allocation strategy can provide every isolation property desired by a tenant or network provider [72].

In this chapter, we present Blender, a model designed to resolve this situation by allowing network providers to deploy multiple isolation schemes across a shared, software-defined network (SDN) infrastructure. As opposed to monolithic approaches, Blender allows providers to pick and choose (as well as create) appropriate models to support the varying needs of the network's tenants. Such flexibility allows network providers to compose contrasting performance isolation schemes with one another and differentiate their cloud offerings through new resource management models.

In designing such a system, we face two key challenges. First, we must ensure that the programming model we offer to network providers is straightforward and usable without overly restricting their ability to express complex isolation policies. In response to this challenge, Blender provides a concise set of abstractions to express a range of allocation schemes that naturally compose with one another, and at run time, the Blender model ensures that multiple schemes will co-exist on a single data center network. Such flexibility allows providers to better support tenant needs and to differentiate their cloud offerings through new resource management and charging models [50, 92].

The second primary challenge is scale. Blender must account for the finite hardware resources available in SDN switches for storing forwarding entries and implementing traffic rate limiters. We overcome these hardware capacity limitations by combining a specialized tenant model with optimization mechanisms to prevent resource exhaustion. These techniques allow Blender's resource allocations to fit within the hardware constraints of modern top-of-rack switches.



(a) Time 0: A network with three tasks to allocate.

(b) Time 1: Task MR_0 installed.

(c) Time 2: Tasks MR_0, Web_1, and Web_2 installed.

Figure 5.1. An example to illustrate how a shared infrastructure might host multiple isolation methods. We first allocate a fixed-capacity MapReduce task followed by two proportional-share Web service tasks. We show how the per-link limits *L* and proportional weights *w* change (in bold) on the physical network after each allocation.

5.1 Requirements for Blending

The choice of isolation model for a particular data center network is a function of its topology, workload, and business model. The disparate set of engineering decisions in previous proposals suggest that a variety of desirable operating points exist, and it seems unlikely that the best answer will be the same for all tenants. Current isolation systems, however, are monolithic: all tenants are treated in an identical fashion. We argue that technical and economic concerns indicate a clear need to deploy multiple isolation models across shared data center networks. For instance, some services may need highly predictable performance, while others may have more flexible requirements, allowing for higher utilization.

As an example, consider how one might support multiple conflicting services on a single network. Figure 5.1 shows a network with four hosts and three tenant requests (which we call *tasks*). Figure 5.1(a) illustrates the network state before any allocations are made—a dumbbell network with 100 Mbps of capacity on each link.

In this scenario, a MapReduce task, MR_0, requires isolation to maintain a predictable level of network performance between hosts a and b. To that end, we provide it with a private, fixed allocation of 60 Mbps. Figure 5.1(b) shows how we might configure the physical network to support MR_0. L_{task_id} denotes the installation of a rate limiter in the network. Note that MR_0's allocation updates the available link capacity on the logical network.

The remaining two tasks both host Web services on hosts c and d. For the sake of discussion, assume that these tasks prefer to proportionally share their capacity such that they receive service according to their relative importance (or payment) and that Web_2 pays three times as much as Web_1.

Figure 5.1(c) shows the result of allocating these two tasks, where w_{task_id} denotes a task's proportional weight allocation. Because MR_0 reserves 60 Mbps on the bottleneck link, the proportional Web tasks share the remaining 40 Mbps in a 3:1 ratio. We update the rate limiters on the chosen links appropriately.

While this example illustrates one form of composition where tasks execute alongside one another, we have identified at least two others: hierarchy and mixture. The second form, hierarchical composition, allows tasks to further subdivide the resources of an existing task. Hierarchical subdivision allows, for example, a provider to partition the network into fixed chunks for separate departments, enabling each department to choose its own internal sharing scheme.

Finally, many of the isolation schemes described in Chapter 2 can be expressed as a mixture of individual features. Thus the third form of composition allows users to express features as modular components that can be re-purposed in other allocation schemes. For instance, Seawall [84] lacks sender coordination, potentially over-running the receiving VM. Such a facility could be provided by distributed rate limiting [74] as implemented in Oktopus [8]. Similarly, Oktopus would benefit from work conservation, a feature common to many other models.

Considering the isolation models and features of recent proposals, we extract the following requirements for a general and practical network virtualization system:

1. Multiple isolation models should be able to co-exist on the same physical network.

2. Each isolation model should be expressed as a modular composition of features that support the tenant environment and simplify model implementations.

3. The system should support reactive, coordinated traffic control to ensure efficient sharing between flows, entities, and tenants.

5.2 The Blender Resource Allocation Model

This section discusses the Blender programming model and how it addresses the requirements outlined in the prior section. A later section (Section 5.4) describes a Blender prototype implementation that cleanly integrates into a full-featured SDN architecture and manipulates the necessary network state.

Blender allows network providers to specify different isolation policies using a small number of high-level abstractions. Utilizing task switching, network providers define isolation policies as separate software modules called allocators. A logically centralized controller runs these modules at the request of tenants and ultimately translates their directives into forwarding and rate limiting state in the network, allowing network providers to create, compose, and execute multiple isolation models across the same physical infrastructure.

In a “blended” network, tenants make requests for Blender to construct tasks, the unit of resource allocation and management that specifies a logical communication pattern. Some tenants may create a single task, such as a long-lived, complex, multi-tiered client service. Other tenants may have multiple tasks, each corresponding to fine-grain application phases, such as the read, shuffle, and write phases of a MapReduce execution.

A tenant specifies a task's isolation requirements by submitting a small, high-level description of the task's communication requirements (including the choice of allocator) to Blender's centralized task controller. Given the arrival of a new task request, the specified allocator analyzes how to (and whether it can) meet the isolation objectives for the task and generates a proposed set of network configuration changes. Blender then determines whether the changes are allowed and feasible, and if so, admits the task, configuring the underlying network to enforce the isolation directives, possibly resulting in the reconfiguration of other tasks' virtual networks[1].

5.2.1 Allocation by Graph

Building upon the task-switching architecture introduced in Chapter 3, Blender grants each task a virtual network in which to operate. When a tenant submits a task request, its allocator computes paths to connect the task's endpoints and creates resource *reservations* to describe how to treat the task's flows. A natural way to make these computations is over a graph; thus Blender represents resources as a logical *graph* of switches/hosts (vertices) and links (edges). Blender uses these graphs to build higher-level abstractions for describing and composing isolation policies.

Each allocation takes a task t and a network graph G as input. The allocator is restricted to using only the resources described in G, even if additional resources may be physically available[2]. The allocator analyzes the graph and computes a new graph representing the task's desired resources: $allocate(G, t, \ldots) \rightarrow G'$. If Blender admits the task, it *removes* the resources (described in Section 5.2.2) in G' from G. Subsequent allocations may either use the remaining resources in $G \backslash G'$ or, in the case of hierarchical composition, those allocated to G'.

[1]This work considers Blender in the context of a trusted environment; providers install allocators that do not sabotage other tasks.

[2]This mechanism may be combined with ACLs or other high-level resource management policies to control resource access.

Table 5.1. An example of composed task allocations and removals and the resulting hieararchy of network graphs.

Time	Task Allocations	Network Graph Hierarchy
0	Initial state	$G_{root,0}$
1	Add t_a at *root*	$G_{root,1} \rightarrow \{G_{a,0}\}$
2	Add t_σ at a	$G_{root,1} \rightarrow \{G_{a,1} \rightarrow \{G_{\sigma,0}\}\}$
3	Add t_β at a	$G_{root,1} \rightarrow \{G_{a,2} \rightarrow \{G_{\sigma,0}, G_{\beta,0}\}\}$
4	Remove t_β at a	$G_{root,1} \rightarrow \{G_{a,1} \rightarrow \{G_{\sigma,0}\}\}$

Hierarchical composition. The ability to pass modified graphs through successive allocations supports on-line hierarchical composition of isolation policies. Thus, the sequence of task arrivals and departures forms a logical hierarchy of network graphs. Table 5.1 depicts an example sequence of graphs in which an initial fixed-rate allocation is sub-divided by two subsequent proportional allocations. To illustrate the changes, we label each network graph $G_{tid,v}$ with the task identifier that created it, *tid*, and a version number v.

Here, a network provider creates the initial graph $G_{root,0}$. A tenant then carves out a fixed-rate virtual network (G_a) by requesting task t_a, which results in a new version of the root graph, $G_{root,1}$, that expresses the remaining, unassigned capacity. Next, the tenant uses a proportional allocator to create two tasks, t_α and t_β, within that virtual network by passing in G_a. This yields two successive versions of G_a ($G_{a,1}$ and $G_{a,2}$) as well as a graph for each of the sub-allocations (G_α and G_β). When task t_β departs, Blender returns its resources to G_a. In this case, G_a returns to the state of $G_{a,1}$, though this may not always be the case (e.g., if task t_α departs instead).

5.2.2 Link Reservations

An allocator implements its policy by *annotating* the provided network graph with link reservations. Table 5.2 describes the graph, link, and reservation elements to which allocators (and extensions) have access. To support the built-in reservation types,

Table 5.2. Allocators manipulate graphs G that contain the following logical elements. Here links and switches are given unique identifiers: *lid* and *sid*.

Name	**Item Definition**
Graphs	$G :=$ {Switches,Links}
Links	$l :=$ {`lid`,`capacity`, `weight`, `state`}
Switches	$s :=$ {`sid`,`state`}
Reservations	$res :=$ {ResType,`lid`,Traffic,args}
ResType	$rt :=$ {`fixed|propo|<user defined>`}
Traffic	$tc :=$ {pattern}

Blender annotates each edge with the current available bandwidth (`capacity`) and the sum of the proportional `weight` currently reserved on the link. Fixed reservations carve out a specific amount of bandwidth from a link that must be less than `capacity`. In contrast, proportional reservations receive bandwidth relative to the link's current weight. Note that it is the allocator's job to compute such weights, which is the principle differentiator between many existing proportional isolation schemes [72, 84].

Allocators create reservations by specifying the type, link, and any additional arguments the reservation may require (e.g., capacity or weight). For example, an allocator may create a proportional reservation with a weight of w and associate it with link l. If this task is admitted, Blender increments l's weight in the input graph by w and sets up policers to rate limit traffic over l to $l.\texttt{capacity} * \frac{w}{l.\texttt{weight}+w}$. Links and switches also contain a `state` annotation, a user-defined dictionary that allows allocators and custom reservation types to maintain additional bookkeeping information across task arrivals and departures.

There are fundamentally two types of bandwidth reservations that one can make in Blender—those that are guaranteed and those that are best effort—and we impose a simple restriction when hierarchically composing allocations: We allow guaranteed allocations to sub-divide a prior guaranteed allocation. Best effort allocations may also sub-divide a guaranteed allocation, but never the opposite. Thus, a best effort allocation

Table 5.3. The API allocators use to create reservations and add annotations. Blender provides an endpoint pair *epp* that describes the task's traffic.

Function	**Description**
createRes(rt, l, args) $\rightarrow res_l$	Create reservation of type *rt* on link l.
addRoute(epp, res_{l_0}, res_{l_1},...)	Route traffic between *epp* over res_{l_i}.

can be used to sub-divide a guaranteed allocation, but any future allocation after the transition to best effort must also be best effort.

5.2.3 Constructing Forwarding Tables

Allocators create reservations and forward traffic using the simple API shown in Table 5.3. The first function, **createRes**, uses the reservation type, the link to apply it to, and any additional arguments the reservation type may require (e.g., capacity or weight) to construct a new reservation. The second, **addRoute**, allows the allocator to specify the path(s) the task's traffic should take through the network.

An allocator is first required to create a reservation for every link it wishes to use (allocators are always free to request zero-capacity, work-conserving fixed allocations to indicate best effort forwarding). Armed with a set of reservations, the allocator can then call **addRoute** to associate the task's traffic between a particular endpoint pair with the associated reservations. Multiple endpoint pairs are free to share the same reservation.

Blender validates the parameters to **addRoute**; the call will fail if the reservations correspond to a disconnected set of links, or if the path(s) constructed by the links do not terminate at the indicated endpoints. If the allocator provides multiple paths, Blender will balance the indicated traffic evenly across them (Section 5.4.2).

5.2.4 Modular Reservation Extensions

Network providers may extend the resource reservation capabilities of Blender and enable new kinds of isolation models by defining their own *reservation extensions*.

Blender provides fixed-rate reservations by default, but some reservation types may require more flexible control over resources. For example, proportional reservations must convert per-link weights *w* to physical limits. Moreover, they must update physical rate limits if any later task decreases the available capacity on a link (as in Figure 5.1). The modular design of Blender reservation extensions allows this flexibility, which we leverage to provide proportional reservations, work conservation, and distributed rate limiting in our prototype (Section 5.4.4).

A reservation extension consists of three primary functions: `res.create`, `res.apply`, and `res.exec`. The Blender controller maintains state for each reservation as the allocator processes the task. Calls to **createRes** cause Blender to call `res.create`; this associates a unique identity with reservation-specific parameters (e.g., the weight or fixed capacity). Blender calls `res.apply` during generation to convert these high-level reservations into physical rate limits. The reservation extension then enables the appropriate policing machinery when it installs the associated forwarding directives to pass task traffic over the link. Static reservation extensions need only implement these two calls.

Reservation extensions that wish to respond to network events may optionally define `res.exec`, an event-driven control loop that manages all instances of the type. These dynamic reservation extensions indicate the events that should trigger the execution of the control loop, which can be both external (e.g., SNMP traps, OpenFlow controller requests, custom device APIs, etc.) or generated by Blender (e.g., a task departure). Inside the control loop, the reservation extension can then inspect the network through whatever API the hardware supports (e.g., SNMP counters) and update its configuration accordingly. For example, proportional reservations use the `res.exec` facility to recompute reservations when tasks arrive or depart. Note that like allocators, Blender reservation extensions are trusted components. Rate-limit modifications must not exceed the available capacity in the task's parent network graph. For example, all proportionally rate-limited tasks sharing a link ensure that their limits do not exceed the link's capacity. Unlike allocators, reservation control loops do not manipulate network graphs. Instead, they interact directly with the underlying network equipment to monitor and modify the state of the network.

5.3 Sample Allocators

This section demonstrates Blender's flexibility in implementing isolation models. We take inspiration from recent data center virtualization proposals and create two allocators: Squid and Jetty. Squid creates fixed, predictable allocations in the spirit of Oktopus [8] and FairCloud's PS-P [72] models, allocating over-subscribed virtual network topologies. Jetty creates proportional allocations in the fashion of Seawall [84] and PS-L [72], providing isolation at the link level.

We describe each allocators' parameters and basic allocation strategy in the following sections. Our descriptions focus on our adaptation of the models and their interaction with the Blender framework. We refer the reader to the original proposals of these models for additional details of their operation. Finally, following the allocator descriptions, we discuss how they differ from their inspirations in Section 5.3.3.

5.3.1 Squid: Predictable Virtual Networks

Squid mimics the data center isolation model provided by Oktopus [8] and PS-P [72], which allows tenants to request virtual networks that emulate non-blocking switches. Each switch i connects n_i VMs with bisection bandwidth b_i. The switch may be further connected to other virtual switches based on an over-subscription factor O. Thus, a virtual switch must have uplink capacity $(b_i \cdot n_i)/O$ to each other switch.

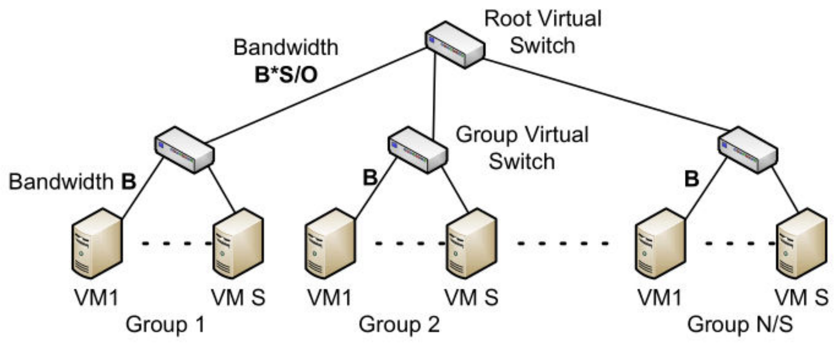


Figure 5.2. A depiction of an Oktopus virtual network, where S is the size of each cluster group, B is the bandwidth of a cluster's virtual non-blocking switch, and O is the over-subscription factor for inter-cluster links. Figure from [8].

Like Oktopus (pictured in Figure 5.2), we assume the input graph is a singly-rooted, multi-level tree.

A complete Squid allocation request is specified as $(G, \texttt{squid}, C_{VMs}, j, n, b, O)$. This request will cause the Squid allocator to embed j virtual switches, each connecting n VMs with links of capacity b, in the input network graph G. Squid will arrange the virtual switches to have an over-subscription ratio of O. The allocator is free to choose the $n \cdot j$ endpoints from the provided $C_{VMs} \in G$ candidates.

Squid uses a first-fit strategy to find feasible virtual switch (cluster) placements. It uses a depth-first recursive algorithm that attempts to place each cluster as low in the topology as possible, minimizing the number of links traffic crosses. For each switch in the topology, Squid records the number of clusters that can be placed beneath it. To determine if a given cluster of size n_i can be placed under a switch, Squid first checks the links l in G for sufficient capacity b to connect a cluster. It then ensures that each found cluster has capacity $\frac{b_i * n_i}{O}$ to every other feasible cluster. If sufficient capacity does not exist (or if too few candidate endpoints were supplied), the Squid allocator rejects the task.

If successful, this process returns a subtree under which all clusters fit. Squid then generates reservations with a top-down traversal of this subgraph, creating a fixed reservation for each link. The reservation capacity is the sum of the connectivity requirements of all clusters under this link. To place the traffic, Squid computes all-pairs shortest path between the selected VMs, calling **addRoute** for each included endpoint pair.

5.3.2 Jetty: Proportional Tenant Allocations

Jetty provides proportional link capacity sharing similar to PS-L [72] at the granularity of tenants (in contrast to Seawall's entity-based proportional shares [84]). For each tenant Jetty ensures a proportional share of capacity on each physical network link the entity uses. While we simplify our discussion of Jetty by assuming a singly-rooted, multi-level tree topology, it may be trivially extended to multi-rooted topologies. A Jetty task allocation request consists of $(G, \texttt{jetty}, C_{VMs}, n, w)$.

Jetty begins by finding the n endpoints in C_{VMs} that are best connected to the network core. It calculates the bottleneck bandwidth on the shortest path from the top-level core switch to each endpoint and then chooses the top-n highest-capacity endpoints. Unlike Squid, Jetty never rejects requests during allocation (assuming $C_{VMs} \in G$ contains at least n endpoints). Like Squid, Jetty notes the lowest level root switch under which all VMs connect, and it creates reservations in a similar top-down process. Each reservation is proportional and uses the same weight w. Jetty then computes all-pairs shortest paths, and calls **addRoute** for each endpoint pair.

5.3.3 Enforcement

It is instructive to discuss how Squid and Jetty differ from their inspirations, which install traffic policers only on sending VMs. Our allocators use in-network traffic policers in addition to end host rate limiting. This decision allows Squid and Jetty to collapse the collection of tenant traffic across each reservation and eliminates the need for the tightly coupled VM sending rate coordination found in Oktopus. With the appropriate optimizations (Section 5.4.3), under our scheme the allocators create at most one reservation per link per tenant, meaning a 48-port switch connecting hosts with 8 VMs would require at most $2*8*48 = 768$ rate policers (assuming every VM is in a different tenant). This policer count is well within the hardware limitations of modern top-of-rack switches; for example Cisco's 4948 E/F top-of-rack switch supports as many as 8K policers [17].

Our use of in-network limiting also simplifies our ability to augment reservations with dynamic rate coordination. By default, neither Jetty nor Squid provide work conservation across tenants. We design a dynamic reservation extension to add work conservation (a feature not available in Oktopus) to both allocators in Section 5.4.4.

5.4 Prototype Implementation

This section describes an OpenFlow-based implementation of Blender using the task switching architecture introduced in Chapter 3. In the OpenFlow model [55], switches are simple forwarding elements whose forwarding tables are populated by an intelligent, logically-centralized controller. While we have not modified our OpenFlow switches, we are leveraging HP Labs extensions to our HP ProCurve switches. The extensions allow an OpenFlow controller to define limiters on a switch and add a rate limiting `action` to OpenFlow rules that reference those limiters. This interface provides the necessary fine-grained control over switch packet forwarding and rate limiting.

5.4.1 Task Controller and Rule Installation

The task controller is implemented in Python as a module for NOX [34], an open-source OpenFlow controller. It maintains all of the state for Blender, including

Table 5.4. Lines of code in our Python implementation of the Blender framework.

Component	**Lines of code**
Blender task controller	2248
Squid allocator	559
Jetty allocator	85
Work conserving extension	191
Distributed rate limiting extension	232
Total	3315

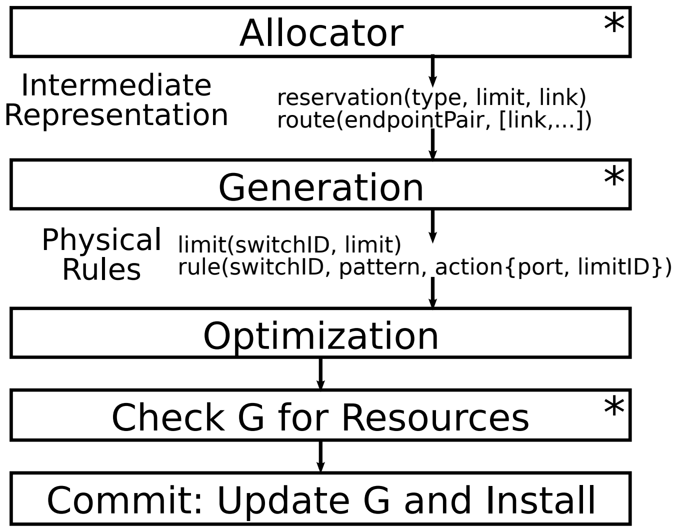

Allocator *
Intermediate Representation
reservation(type, limit, link)
route(endpointPair, [link,...])
Generation *
Physical Rules
limit(switchID, limit)
rule(switchID, pattern, action{port, limitID})
Optimization
Check G for Resources *
Commit: Update G and Install

Figure 5.3. After allocation, compilation proceeds to convert an intermediate representation of abstract limits and routes into a concrete set of rules. Any stage with an * can abort the task before it commits.

the physical network representation, allocated network graphs, connections to end host clients, and the limiters and rules installed in the switches and end systems. The remaining components of the system are implemented as modules within the Blender task controller, including the library of reservation extensions and provider-installed allocators. Table 5.4 shows the code size of our Python prototype, breaking out allocators and our dynamic reservation drivers. Useful allocators and drivers may be written in less than a few hundred lines of code.

Blender needs to ensure two important forms of task-wise concurrency control. First, allocators must receive a consistent view of the input network graph. A task should not observe other allocations and should execute only if it can receive all its resources. Second, a task's rules must be fully installed before sending traffic. This requirement provides *per-task* forwarding consistency; all packets of a given task obey one set of forwarding and rate limiting rules at a time.

Blender provides these semantics by processing a task request as a transaction. Each allocation involves creating reservations, generating switch rules, optimizing the rule count (Section 5.4.3), and installing rules into the network. We call the first three steps *task compilation* as a task's routes and resource reservations must be converted into SDN-compatible forwarding rules. Figure 5.3 shows this sequence of events as Blender handles a task request.

Stages marked with an asterisk may abort the transaction before any changes are made to the physical network. For instance, SDN switches have limited storage capacity in their forwarding tables, and the task controller ensures that all network elements have sufficient space for the task's optimized rules. At this point Blender commits updated annotations to the task's input graph to reflect the task's resource claims. Blender commits tasks in the order it receives them. If two allocations attempt to use the same resources, one will ultimately commit before the other, causing the second to roll-back and either try again or be rejected, thus preventing tasks from executing without receiving their full set of resources.

Finally, after successfully committing updates to the available network resources, Blender pushes the task's rules and rate limits into the network. To ensure the task's forwarding rules are consistent across the network, we wait until the task's network state is fully installed in all devices before notifying the requesting application that its network is available and ready for use.

An additional challenge specific to our testbed environment is ensuring that all rules reside in fast-path hardware lookup tables. Our switches can only install a limited number of rules into their TCAM in a short period of time, and additional rules are silently added to a slower software table. To prevent these slow software rules, we rate limit our rule installation and periodically check for and move any rules that are found to be in the software table. We have empirically determined that issuing eight rules per second and checking for software rules every ten seconds works well for our switches. To cope with failures, the task server records the set of rules for each switch, and it will re-dispatch rules to any failed switch when it reconnects.

5.4.2 End-system Shim Layer

Tenants interact with Blender by submitting task requests. They do so by communicating with a local Blender shim that is running in user-space on the VM. The shim maintains a communication channel with the task controller. Upon connecting to the task controller, the shim transmits a unique identifier on behalf of the host it represents. The controller considers this value the endpoint identifier and associates it with the physical location of the VM. While this process binds the VM to a physical host in our prototype, it would be straightforward to move the shim into a hypervisor or adopt recent proposals for data center address virtualization [62].

Tenants submit a network task request to the local shim, which relays it to the Blender task controller. After the controller installs rules and limiters at the switches, the controller replies to each shim in the task. The reply contains a set of type of service (TOS) bits and an optional rate limit for each destination in the task. Upon receiving this information, the shim manipulates `iptables` to mark outgoing packets with TOS bits and `tc` to rate limit flows. For destinations with multiple paths, the shim installs TOS marking rules that split traffic evenly among the available paths at a flow-level granularity. To the controller, the shim appears no different from a switch; the controller will re-install rules/limits if the VM reconnects after failure. If a task does not pass the allocation phase, the shim will receive a callback from the controller indicating this failure.

5.4.3 Rule Optimization

OpenFlow rules eventually reside in switch memory, a limited resource. Our testbed switches store rules in a TCAM, which allows wildcard matching via “don’t care” entries. TCAM sizes are limited due to cost and power requirements, so Blender’s rule optimization phase tries to conserve space in these switch tables by combining rules into a single wildcarded TCAM entry, when appropriate. The optimizations improve the scalability of the system by allowing more (or larger) tasks to be installed.

Blender’s optimizer leverages two abstractions to combine forwarding rules: our task-based allocation strategy and a hierarchical network identifier space, as enabled by several recently proposed virtualization systems [32, 62, 63]. For each switch port in use by a task, it uses wildcarding to collapse all of the port’s rules into a match on the longest-prefix destination and TOS field of the rule set. The optimizer is space-efficient and produces only one forwarding rule per switch port per task. Unfortunately, our switches do not respect OpenFlow’s rule priority field, making longest prefix matching, and the use of this optimizer in our testbed, impossible.

We implemented a second optimizer that is simpler and less space efficient, but does not require rule priorities, making it deployable on our switches. It uses a simple heuristic that attempts to wildcard the source address of rules at each switch. The algorithm finds the set of rules that share the same destination, TOS field, limiter, and output port. If the source address is identical for the entire set, the optimizer will collapse them into one rule with a wild card for the source address. Once this optimization occurs, the Blender task controller must not admit any future rules that match this wildcard. We use this optimizer in our testbed evaluations.

5.4.4 Reservation Extensions

While Blender supports fixed and proportional link reservations by default, network providers are free to extend Blender with their own reservation types. Moreover, such reservation extensions are modular; a link can support many reservations of different types (e.g., composing work conserving reservations with fixed rate guarantees). We leverage this facility in our prototype to design reservation extensions for work conservation and distributed rate limits. Allocators combine these reservation types with existing, fixed reservations on the desired link(s). For instance, a work-conserving version of the Squid allocator simply pairs a work conservation reservation with each fixed reservation.

Work conservation. Work conservation allows a task to contribute its unused portion of a reservation on a link to an excess capacity pool. If other tasks on the same link are operating below their reserved capacities, capacity-hungry tasks can draw capacity from this pool. Note that the work conservation reservation extension described here is implemented by querying for traffic demands and modifying the rate limits of the hardware policers that are already associated with the tasks' reservations. It does not require weighted fair queueing or any other explicitly work-conserving hardware mechanisms. This design also illustrates the flexibility of reservation extensions, as the excess capacity pool may be shared among all reservations on the link, regardless of their allocator, or it may be scoped to a subset of the link's reservations.

Distributed Rate Limiting. DRL provides a mechanism whereby multiple senders coordinate their sending rates to ensure that their aggregate traffic rate is below a maximum global rate limit, without statically limiting their rates. The DRL reservation extension periodically checks the rates, across multiple links, of a coordinated set of traffic and adjusts link-local rate limits in proportion to the traffic's demand across those links. The application of DRL is useful in a situation in which a task would like to control the aggregate sending rate across paths that may not share any common hops. For example, in a multi-tiered, distributed service, a tenant may want to limit the global rate at which one tier can transmit to another.

5.5 Evaluation

Our evaluation explores Blender's ability to run multiple, composed isolation models on the same network. We measure the overheads of task allocation and installation and show that, even with the limitations of current OpenFlow prototypes, Blender can react to traffic observations and provide effective isolation between tasks. We emulate a task-intensive workload over both Squid and Jetty, studying the impact of work conservation on their performance. Finally, we characterize the behavior of two reservation extensions. These experiments allow us to highlight Blender's features, quantify Blender's resource usage, and demonstrate its practicality for networks containing thousands of switches.

5.5.1 Physical Testbed

We explore Blender using a three-level fat tree [2] built from six 48-port HP ProCurve 6600-series switches running the K.14.87o OpenFlow firmware. The switches can store 1500 OpenFlow rules and contain 256 hardware rate policers. We use VLAN separation to divide each physical switch into eight six-port mini-switches. We arrange the mini-switches into a $k = 6$ fat tree [2] in which there are six pods, each with nine hosts, three edge mini-switches, and three aggregation mini-switches. Note that while our prototype runs across the full fat tree topology, we limit our topology to a single core mini-switch to effectively reproduce topologies used in experiments in the Oktopus [8] and Seawall papers [84].

The switching fabric carries traffic for 50 hosts, each of which contains an Intel Xeon X3210 and 4 GB of main memory. Every host uses two gigabit Ethernet interfaces; the first connects to a control network for system administration and interfacing with the task server, and the second interface connects to one of the edge-level, Blender-enabled mini-switches. The hosts each house four Linux-KVM virtual machines, for a total of 200 VMs.

5.5.2 Isolation

We demonstrate Blender's ability to combine multiple isolation models on a shared physical network by hierarchically composing allocators, as described in Section 5.2.1. This experiment uses the Squid allocator to isolate sets of Jetty tasks from one another. We first use Squid to create two virtual networks of fixed capacity: 600 Mbps (NetGraph_1) and 300 Mbps (NetGraph_2). Within each virtual network, we allocate two proportional Jetty tasks. We have engineered the task requests to ensure that all six allocations share a common path in the core of the physical network[3]. We then start traffic flows within the four Jetty tasks at various times.

Figure 5.4 shows the rate utilization across the shared physical path. The three sub-figures show total utilization, traffic for NetGraph_1, and traffic for NetGraph_2, from top to bottom, respectively. Initially, a single Jetty task of weight 3 is executing on NetGraph_1. At time 15, a Jetty task of weight 1 begins executing on NetGraph_2. Fifteen seconds later, a second Jetty task of weight 1 arrives on NetGraph_2, causing them to each share 50% of NetGraph_2's 300 Mbps capacity. At time 45, a Jetty task of weight 1 starts on NetGraph_1, triggering NetGraph_1's capacity to be reallocated in a 3:1 ratio. At subsequent 15-second intervals, one of the remaining tasks finishes and departs.

Notice that each pair of Jetty tasks performs as if they were on their own network,

[3]This was done by setting C_{VMs} to control the placement of VMs.

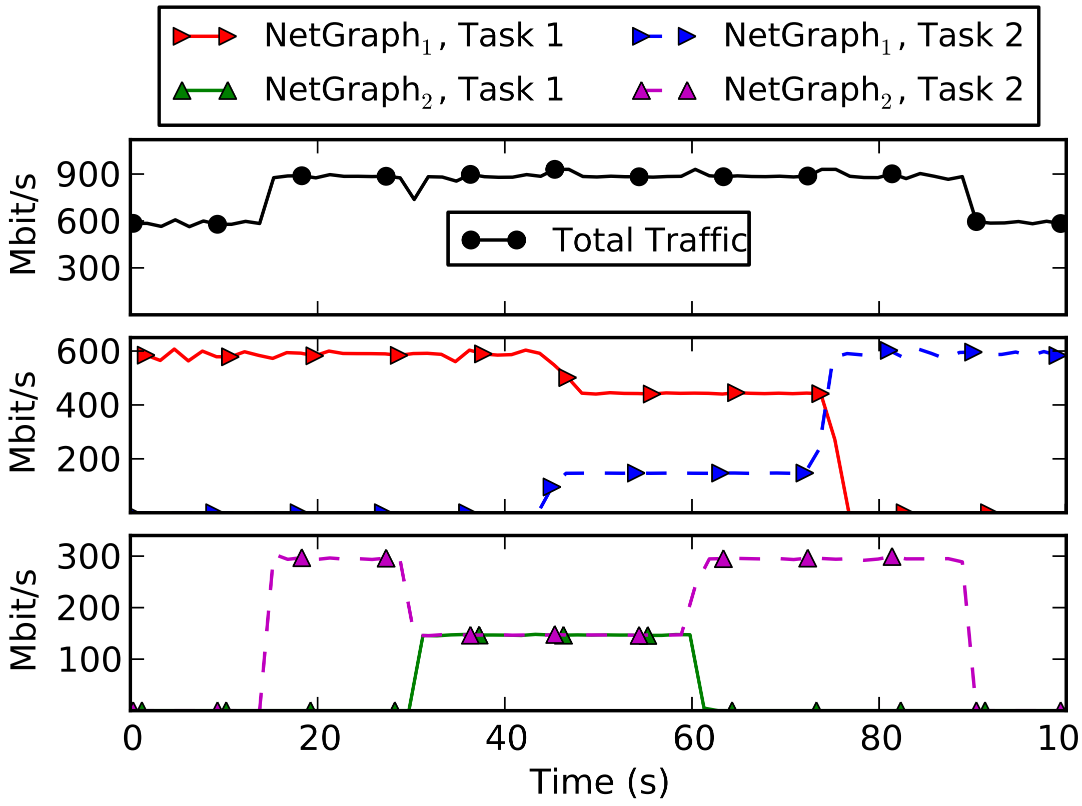


Figure 5.4. A composition of multiple isolation policies over a shared physical path. The three figures show total utilization, traffic for a 600-Mbps network slice, and traffic for a 300-Mbps network slice, from top to bottom, respectively. Each slice's events do not impact the tasks operating in the other.

responding only to Jetty task arrivals on their Squid virtual network. Moreover, the combined traffic never exceeds the sum of the capacities of the Squid virtual networks.

5.5.3 Task Throughput

Next, we evaluate the time for Blender to complete a set of 100 tasks, using both Jetty and Squid, in a similar fashion to the evaluation in [8]. Each of the 100 tasks requests a number of VMs drawn from an exponential distribution with a mean of 18. Each VM in the task chooses one destination among the other VMs in the task and sends 1500 MB to it. The task is considered complete when every VM has completed its transfer. A task dispatcher requests tasks to Blender, and each requested task is either accepted,

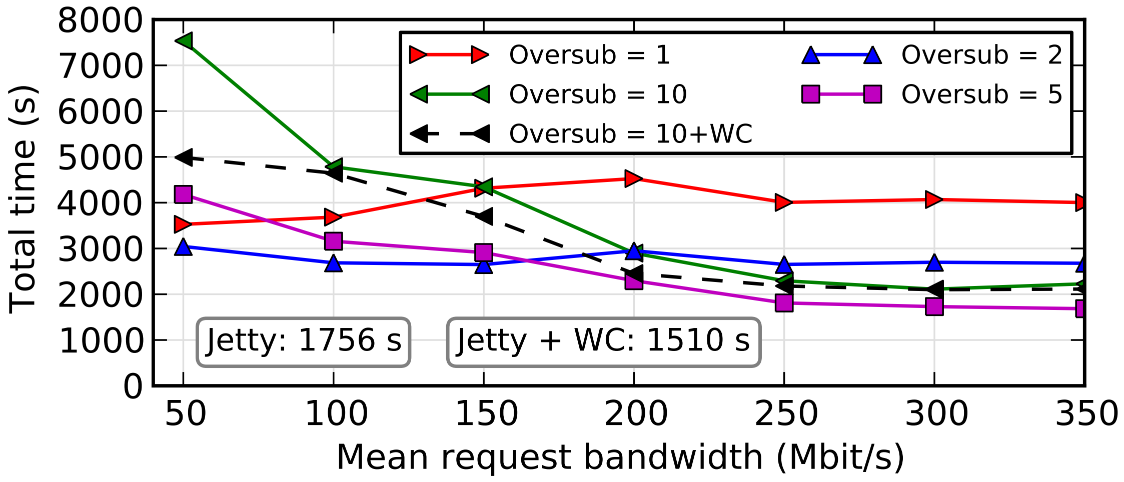


Figure 5.5. The time it takes Jetty and Squid to complete a set of 100 tasks as we vary the task request parameters.

in which case the task begins running, or it is rejected due to insufficient resources (lack of capacity or available VMs). The task dispatcher continues to request any rejected tasks until they are all completed. We measure the time between the first task beginning and the final task ending. Without a system like Blender, such a comparison would be difficult to perform.

For Jetty, each VM in a task is equally likely to send to any other VM in the task, and all tasks request equal weights for their paths. For Squid, the parameter selection process is similar to that in the evaluation of Oktopus [8]. Because Squid sub-divides its tasks into clusters, we assign VM destinations according to the oversubscription factor O such that the likelihood of a VM choosing an inter-cluster destination is $\frac{1}{O}$ for $O > 1$ and uniform likelihood for $O = 1$. Finally, Squid tasks draw their bandwidth request value from an exponential distribution whose mean we vary.

Figure 5.5 shows the time it takes each configuration to complete the set of 100 tasks. We plot the completion time for Squid oversubscription factors of 1, 2, 5, 10, and 10 with work conservation enabled. We also show Jetty with and without work conservation. Jetty has no notion of bandwidth or oversubscription factor, so we present

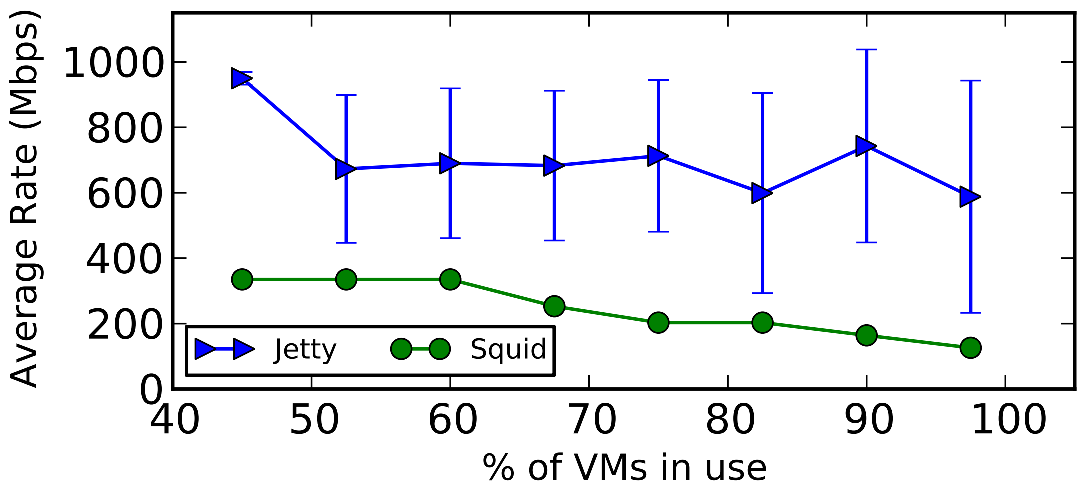


Figure 5.6. The mean and standard deviation (bars) of the bandwidth between the VMs of a 15-node experimental task with a varying number of background tasks.

Jetty's completion times separately on the plotted results.

Because it shares capacity evenly among concurrently executing tasks, Jetty is only constrained by the number of available VMs. As a result, it tends to keep network utilization high, and it performs relatively well, particularly with work conservation enabled. For low mean bandwidths, Squid tends to run out of VMs before it exhausts the available network capacity, leading to poor performance for larger oversubscription factors. Enabling work conservation provides a significant benefit in this region due to improved network utilization. As we increase the mean bandwidth, oversubscription becomes more beneficial due to capacity, rather than VMs, becoming the constraining resource. For the number of VMs in our network, an oversubscription factor of 5 appears to perform the best for Squid tasks.

5.5.4 Allocator bandwidth and variability

In the previous experiment, Jetty achieves a higher overall throughput than Squid. However, the increase in throughput comes at a cost with respect to Squid in the form of variance. To illustrate this effect, we adjust the load on the network by installing

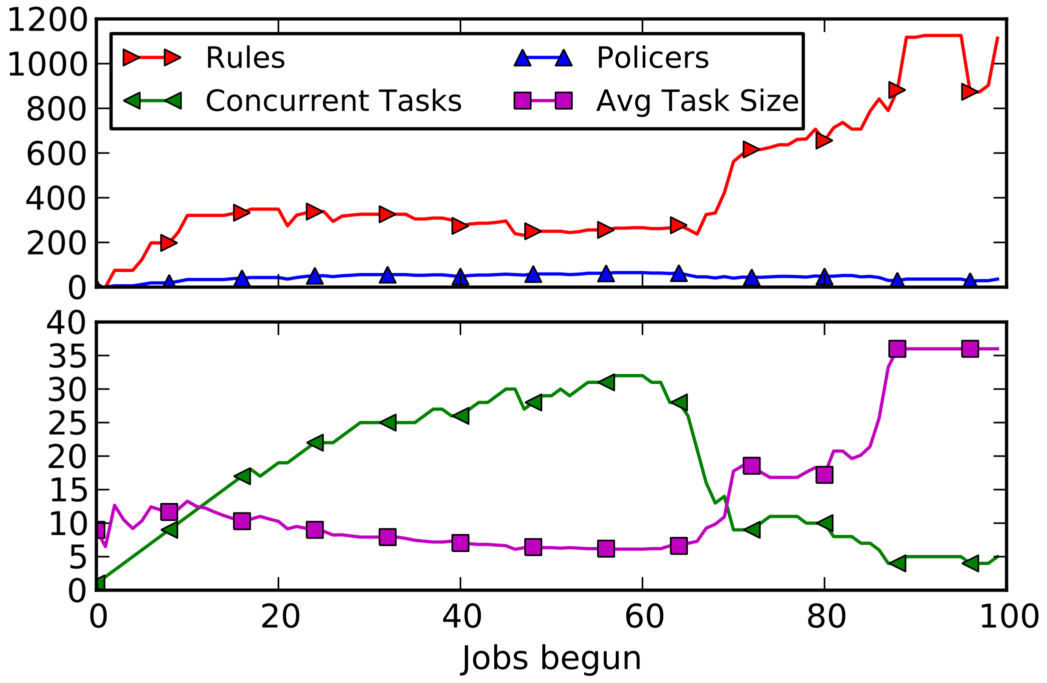


Figure 5.7. Rule and policer count, along with the number and average size of concurrently executing tasks for a switch during the Squid $Mean = 150$, $Oversub = 5$ run described in Figure 5.5.

a varying number of background Jetty tasks. Next, we install one experimental task and measure the capacity between all pairs of its VMs. Figure 5.6 plots the mean and standard deviation of the experimental task as we vary the total number of VMs in use. As expected, provisioning the experimental task with the Jetty allocator yields a higher average capacity than Squid, which has a substantially lower variance. This experiment demonstrates that if given the opportunity to choose, different tenants would benefit from selecting an allocator that matches their applications' bandwidth and performance variability requirements.

5.5.5 Hardware Resources

Switches have a finite hardware capacity, which constrains their ability to store rules and police traffic. These limitations affect the number of concurrent tasks Blender

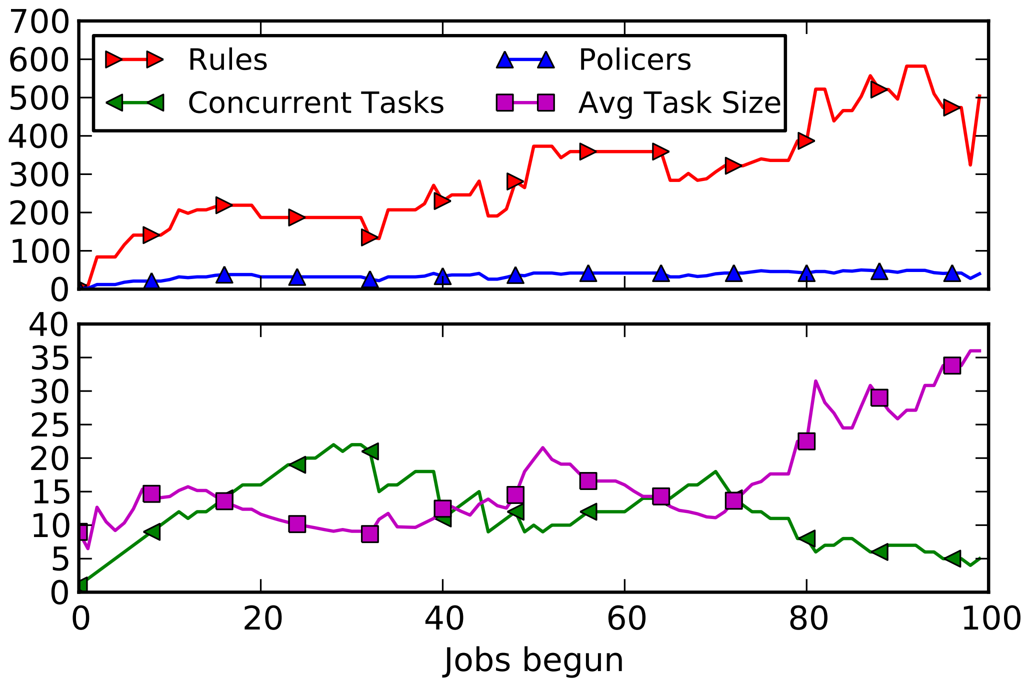


Figure 5.8. Rule and policer count, along with the number and average size of concurrently executing tasks for a switch during the Jetty run described in Figure 5.5.

can support, which is a complex problem. As the number of rules and policers needed by a switch depends on task count, task size, network topology, and allocation strategy. For Squid and Jetty tasks, policer usage scales according to the number of executing tasks, while rule usage more closely follows task size.

Figure 5.7 illustrates the rule and policer counts for a single switch, along with the number of concurrently executing tasks and average task size, during the execution of the Squid $Mean = 150, Oversub = 5$ run depicted in Figure 5.5. We selected the switch with the maximum rule count during the run. The other switches followed a similar pattern. As we described in Section 5.4.3, we use a sub-optimal rule optimizer due to our switches not implementing rule priority. With our better optimizer, the rule count would be equal to that of the policer count, which is a significant decrease.

Over the course of the experiment, we do not prescribe the order in which tasks

Table 5.5. The time and space requirements for a simple two-node task, a Squid task of size 4, and a Squid task of size 36.

Allocator	Simple	Squid	
VM count	2	4	36
Action	**Time**		
Allocation	0.407 ms	2.31 ms	86.3 ms
Rule generation	0.327 ms	2.17 ms	280.0 ms
Rule optimization	0.0898 ms	0.559 ms	64.8 ms
Task construction	0.0262 ms	0.132 ms	16.4 ms
Rule installation	229.0 ms	805.0 ms	15100.0 ms
Resource	**Count**		
Rules (pre-opt)	6	44	5496
Rules (post-opt)	6	20	468
Rate Limiters	0	[4-12]	36

execute. Our task dispatcher requests tasks one at a time, cycling though all outstanding tasks until it finds one for which the system has sufficient resources. As the lower graph indicates, this scheme tends to initially execute smaller tasks, as they are more likely to find the resources they desire as tasks enter and leave. Around the time the 60th job begins executing, the switch reaches a peak of 65 allocated policers, which corresponds to the maximum concurrently-executing task count of 32. As the system shifts to larger task sizes, the concurrent task count decreases. The transition to larger tasks leads to an inflection point in the graph, beyond which the usage of policers decreases and the number of rules increases.

Figure 5.8 shows the utilization of the same resources during the execution of Jetty tasks. The trends for Jetty are similar to that of Squid, though less pronounced, due to Jetty not being constrained by available capacity.

Another important resource in any SDN is the controller. As described in Section 5.4.4, Blender allows for reservation extensions to execute code on the controller in response to dynamic network conditions. While reservation extension execution may sound like it would tax the controller, the reservation extension programming model

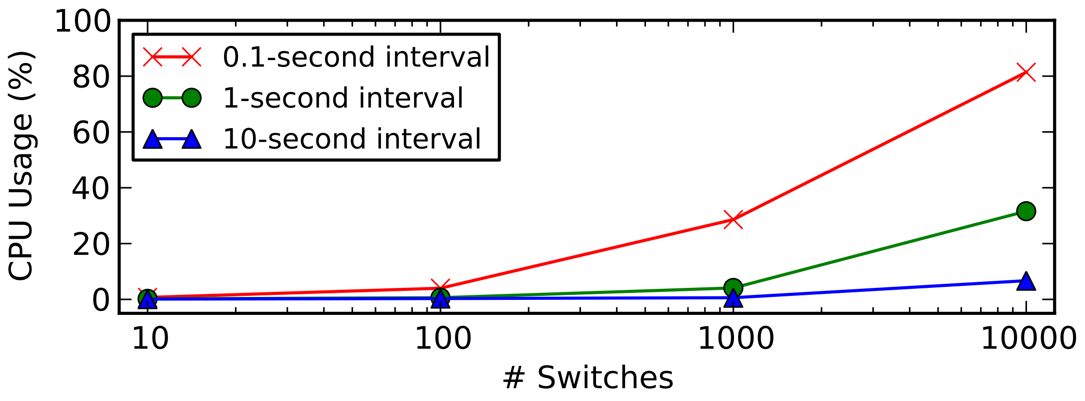


Figure 5.9. Work conservation's CPU utilization at the controller for a variety of switch counts and control loop intervals.

naturally enables extensions that are light-weight, with a tunable control loop interval to trade off accuracy and resource usage.

As an example, our work conservation extension executes a control loop in which it periodically requests switch statistics and reassigns rate limits. Figure 5.9 shows the processing requirement of the work conservation extension at the controller, varying the control loop interval and the number of switches.

5.5.6 Task Installation Overheads

Data centers must set up many tenants, stressing the scalability of the allocation architecture. We explore the overheads Blender incurs to compile and install network tasks of varying complexity. These experiments measure overheads for a strawman two-node task with a single bi-directional link as well as two Squid tasks (our most expensive allocator) of sizes 4 and 36. Table 5.5 breaks down timings by component, and lists the number of rules and rate limiters used per task.

The takeaway from these numbers is that even for a relatively large task (Squid 36)[4], total compilation time (without installation) is roughly 500 ms. The time to install

[4]Public cloud offerings typically limit the number of VMs a user may instantiate. Users of EC2, for example, must request special permission to use more than 20 VMs.

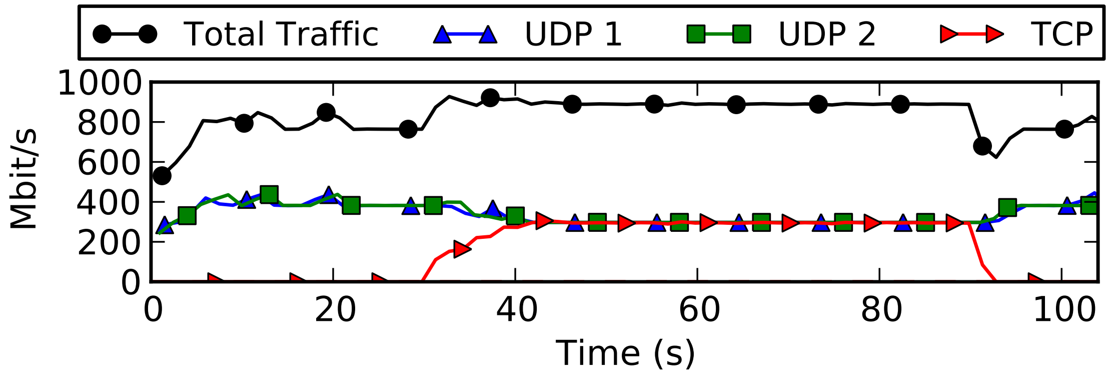


Figure 5.10. The work conservation dynamic module controlling the traffic of three senders across a shared physical path. Each sender has a fixed reservation of 300 Mbps, which it contributes to work-conservation pool for a total of 900 Mbps. The work conservation module divides the pool among the set of active senders, which varies over time.

the rules is over 15 seconds (recall we can only install 8 rules / switch / second). This delay illustrates the importance of rule optimization; our simple optimizer (Section 5.4.3) improved rule count by an order of magnitude.

5.5.7 Dynamic Policies

To exhibit the functionality of dynamic isolation policies and custom reservation drivers described in Section 5.4.4, we evaluate two dynamic reservation modules: work conservation and Distributed Rate Limiting (DRL) [74].

Work conservation. To demonstrate work conservation, we configure and install three tasks, each of which reserves 300 Mbps across a shared path and contributes its reservation to a work-conserving pool. Figure 5.10 depicts the experiment's result. Initially, two of the tasks send UDP datagrams as quickly as possible across the shared path. Due to the third task being idle, the UDP senders draw from the excess work-conservation pool and each send at just over 400 Mbps. After 30 seconds, the third task begins sending TCP traffic over the shared path. Despite competing with UDP, the work conservation module ensures that the third task's TCP traffic receives its 300-Mbps reser-

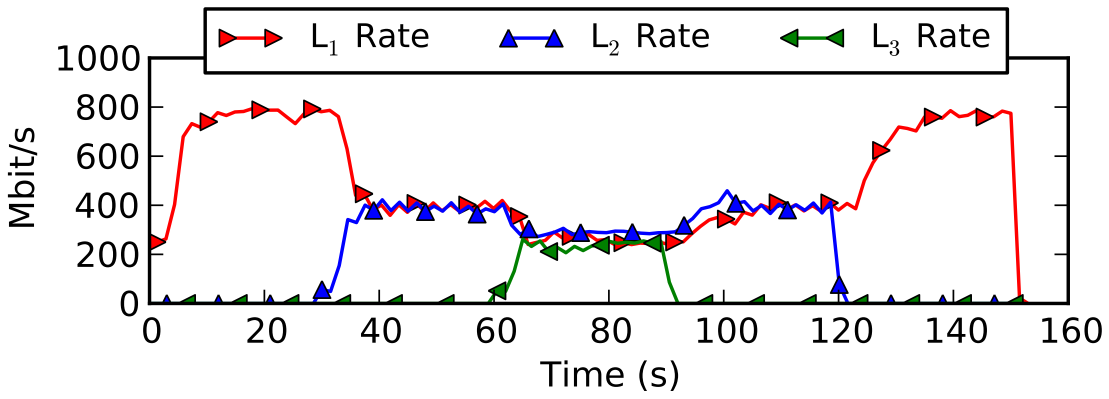


Figure 5.11. The Distributed Rate Limiting (DRL) module enforcing an 800-Mbps global rate limit across three physically-disjoint network paths. The plot shows the outgoing rates of the three limiters, L_1, ..., L_3, that are responsible for policing the three paths. As senders begin and end their transmissions, the total traffic rate remains below the global 800-Mbps limit.

vation. At 90 seconds, the TCP flow completes, and the UDP senders are again given the excess pool capacity.

Distributed Rate Limiting. To test our DRL module, we construct a task in which a set of three sending VMs transmit to three receiving VMs across three fully-disjoint paths. For each path, the sender creates a fixed 900-Mbps reservation along the path to its corresponding receiver. We then configure the edge-switches of all three senders to participate in a DRL reservation with a global rate limit of 800 Mbps.

Figure 5.11 shows the traffic departure rates from the edge-switch rate limiters associated with the traffic for each of the three senders. Every 30 seconds, one sender begins or ends its transmission, causing the DRL module to reallocate the 800-Mbps budget amongst the active senders. Without DRL, we would see each sender fully utilizing its 900-Mbps reservation, however with DRL enabled, the aggregate sending rate of all three senders remains below the global limit.

5.6 Summary

This chapter introduced Blender, a programming model that enables data center providers to author and compose multiple isolation models on the same network. Blender's primary contributions can be summarized as the following:

- **Blender performance isolation model:** Blender is a modular model that allows network providers to simultaneously execute multiple isolation models, both side-by-side and hierarchically, on shared physical resources. Providers define an allocation scheme as a composition of *reservations* made across a logical graph representing the network. Some reservation types may provide fixed capacity guarantees while others could enforce proportional rate limits. Blender enables mixing of standard isolation functionality, such as work conservation, to augment a pre-existing static allocation policy.

- **Blender network architecture:** Blender provides this flexibility while ensuring a consistent and scalable forwarding infrastructure. It multiplexes network resources at the granularity of network *tasks*, which represent an entire service (e.g., a tenant) or individual application phases that have specific network demands. Transactional task allocation ensures atomic and isolated changes to network state.

- **Implementation and evaluation:** We illustrate these concepts through an OpenFlow based [55] prototype. We demonstrate the ability to author, compose, and execute different isolation policies, including the fixed and proportional bandwidth sharing strategies found in recent work [8, 72, 84]. Blender leverages switch-based traffic policers to simplify resource allocation and dynamic traffic control; we describe the resource requirements for task-based allocations and show that for realistic resource allocation strategies, per-switch rule and policer counts are bounded by $O(\text{numTasks} * \text{portCount})$. Even without those assumptions, our tenant-churn experiments on a 50-node, 200-VM, fat-tree testbed easily fit within the resource constraints of HP's prototype OpenFlow switches.

While Blender can express many recently proposed isolation models and easily enables the addition of new features to them (e.g., work conservation), it focuses exclusively on network performance isolation. The next chapter introduces Omakase, extensions to the Blender model for providing tenants with an "app store" of other useful network features. The Omakase model enables tenants to deploy general functionality such as failure recovery or flow scheduling directly within the network infrastructure.

Acknowledgements

Chapter 5 contains material that has been submitted for publication in the Proceedings of the ACM Symposium on Cloud Computing (SOCC), 2013. "Blender: Mixing Data Center Network Isolation Models". Webb, Kevin; Yocum, Ken; Snoeren, Alex. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Omakase: An App Store Model for the Data Center Network

This chapter describes Omakase, a model for writing, deploying, and managing applications that control the underlying network hardware used by tenants multiplexed across a shared cloud infrastructure. Just as the POSIX [95] family of standards maintain compatibility between operating systems, Omakase defines a similar set of standard interfaces that allow cloud providers to author a variety of network applications.

The Omakase model allows cloud service providers to construct Apps that provide distinct pieces of network functionality to their tenants. Like the smartphone apps from which they were inspired, Apps rely on specific facilities of the underlying platform to support their implementation. On phones, these features are generally provided by sharable sensors that are disjoint from one another (e.g., the accelerometer can be simultaneously read by multiple apps and has no impact on the GPS). In contrast, many resources in the underlying fabric of a cloud network are not intrinsically sharable. Thus, one of the primary challenges for Omakase in presenting a smartphone-like app abstraction is detecting and resolving App feature conflicts.

Just as few smartphone users have identical app configurations on their phones, cloud computing tenants are likely to vary greatly in the set of functionality they require from the network. Furthermore, smartphone users generally don't write their own apps, they prefer to choose from among a set that has been approved by their provider, and we believe this preference will persist in the cloud. As an example, consider three tenants that share the physical data center network of a public cloud computing platform: a tenant managing a three-tiered Web service, a tenant executing a bulk data processing job (e.g., MapReduce or Hadoop), and a tenant hosting back-end business logic in the cloud. While all three tenants want some form of performance isolation, each would benefit from a different set of supplemental Apps to support their execution:

- **Three-tiered Web service:** The Web service requires external routing (to interface with customers outside the data center), load balancing (to direct incoming service requests), and bounded latency for communication between tiers (to meet customer performance objectives).
- **Bulk data processing:** The bulk processing job wishes to maximize its throughput with multiple links and a traffic-aware App for assigning flows to paths.
- **Back-end business logic:** The back-end service needs external routing with virtual private networking (VPN) capability (to interface with the business's other systems and intrusion detection (to prevent unauthorized access).

The Omakase model lets the provider export these types of functionality to tenants, who need only make simple requests to incorporate it into their virtual network. In Omakase, a tenant's Apps have the potential to conflict with one another if they attempt to use the same features (e.g., rate limiting) in a shared region of the network. To accommodate this possibility, Omakase additionally accounts for the location in which the App intends to execute and requires that cloud providers enable sensible, common-case conflict resolutions. This section describes the expectations placed on cloud providers for developing Apps, Omakase's facilities for detecting and resolving App conflicts, and the process by which tenants request Apps to run on their behalf.

6.1 Developing Apps

Omakase provides flexibility for tenants to customize their virtual cloud networks while ensuring that cloud operators ultimately retain control over their network. Therefore, we require that the cloud provider, rather than the tenants, construct and supply Apps to ensure that they are not malicious or otherwise abusing the network's resources.

To Omakase, an App represents a collection of functionality that can be chosen by a tenant to operate within a configurable subset of the tenant's virtual network. Apps perform their responsibilities in two phases. First, before a tenant begins executing on the network, Omakase executes a static provisioning routine for each of the tenant's selected Apps. Provisioning allows each App to subscribe to event notifications from network devices and initialize its state for execution, both at the controller and in the network.

After the virtual network and Apps are provisioned, the tenant may begin using the network. Completion of provisioning marks the second phase, in which an App will receive a notification when its subscribed events are triggered. The App may execute an arbitrary callback function in response to an event notification, where the App may choose to change its event subscriptions or update the state of the network. Table 6.1 describes the API available to Apps for subscribing to events and updating network state.

6.1.1 Atoms

Omakase does not prescribe that Apps be implemented in any particular language or controller software. Instead, the distinguishing characteristics of an App are

Table 6.1. Omakase App API for managing event subscriptions and performing network state updates.

Function	**Description**
subscribe(event, location, callback function)	Subscribes to an event, registering a callback.
unsubscribe(event, location)	Unsubscribes from event notifications.
update(atom, location, [atom-specific arguments])	Updates network state, via the specified atom.

determined by the *atoms* it uses to carry out its goals. Atoms abstract the features and functionality of the network, for example, rate limiting, providing traffic statistics, selecting flow paths, modifying packets, etc. Despite many devices implementing and exposing these features differently (e.g., in switch hardware vs. at the hypervisor), Omakase unifies each atom into one abstraction whose App interactions are translated to the appropriate interface for the underlying device.

Omakase requires the network provider to enumerate the set of available atoms and their interface definitions. While providers are free to define their own atoms, Table 6.2 summarizes our default set of Omakase's atoms and their properties. When operators develop an App, they label the App with its required atoms to inform Omakase of how the App will be using the network resources[1]. Labelling an App as 'using an atom' provides two primary benefits. First, it supplies the App with an atom-specific interface, expanding the set of events or network state updates for the App to utilize. Additionally, labelling enables conflict detection during the static App allocation phase.

6.2 Conflicts

In Omakase, a conflict occurs when two or more Apps attempt to modify the network, using the same atom, in a manner that is unsafe without external coordination. To account for the differences in atom behavior, the specific conditions that represent a con-

[1]Atom labelling is analogous to developers labelling Android applications with their desired security permissions.

Table 6.2. A summary of Omakase's atoms and the facilities they provide.

Atom Name	**Conflict Resolution**	**Allows App to . . .**
Create zones	Static	Sub-divide network regions.
Choose routes	Static	Establish VM reachability.
Create rate limits	Dynamic	Add or remove rate limits.
Modify rate limits	Dynamic	Change the enforced rate limit.
Read statistics	N/A (read-only)	Receive network traffic information.
Assign flow paths	Static	Bind a flow to a set of hops.
Intercept traffic	Static	Inspect packet content.
Modify traffic	Static	Re-write packet content.

flict are atom-dependant. Omakase uses spatial *zone* separation to reduce the likelihood of conflicts and performs atom-specific resolution for conflicts that cannot be avoided. Zones are defined hierarchically, as a subset of a graph representing the network's devices and links. Omakase expects tenants to specify in which zone each of their Apps will execute. We provide a special zone creation atom that allows for the definition of new zones. By convention, our implementation restricts this atom to Apps that provide performance isolation (Section 6.3).

Even with zones scoping the use of atoms, multiple Apps may wish to share an atom within a single zone. When such a conflict is discovered, Omakase executes a provider-defined, atom-specific resolution routine. The conflict resolution routine is free to take any information about the state of the network or the tenant's App set into account. Depending on the cloud provider's policies and atom in question, resolution strategies may involve prioritizing one App over another, processing each App sequentially, or coordinating the Apps such that they safely share network resources. For Apps that require coordination, the requirement is enforced in the atom interfaces that they declare.

The Omakase model makes a distinction between two forms of conflicts, static and dynamic conflicts, which correspond to the two phases of App execution. To ensure safe execution, a static conflict is one that must be detected and resolved during the static App provisioning phase. Consider an example in which a tenant requests multiple Apps that assign flows to routes. We leverage atom labelling to detect this condition, and for this example, resolve the conflict by rejecting this (nonsensical) tenant request. With a different atom, for example modifying traffic, Omakase may choose to resolve a conflict by ensuring a sequential ordering on the flow of traffic through the set of conflicting Apps.

For some atoms, the presence of a conflict may depend not only on Apps sharing a zone, but how the network state is managed by the Apps within that zone. For example, if multiple co-located Apps declare that they modify rate limits, whether or not they conflict depends on the set of traffic for which each App intends to modify limits. For atoms that have a additional parameter, like the traffic set in this example, the network provider may opt to resolve their conflicts at runtime, while the Apps are dynamically executing. Dynamic conflicts are detected and resolved by the atom implementation in App `update` calls.

6.3 Prototype Implementation

This section describes a prototype implementation of the Omakase model as an extension to the Blender model introduced in Chapter 5. Our Omakase prototype shares many of the goals and principles behind Blender. Like Blender, Omakase adheres to the task switching philosophy of holistically allocating resources at the unit of tasks. Omakase also maintains the simplicity of Blender's task request specification. In the interest of making Omakase easy for service operators, one of our goals is to put the App configuration burden on the cloud provider, rather than the tenants, whenever possible. We believe that it is unreasonable to expect detailed configuration information from all tenants, and thus tenants need only provide the names of the Apps they wish to run and the zones they want them to run in (if not in all zones).

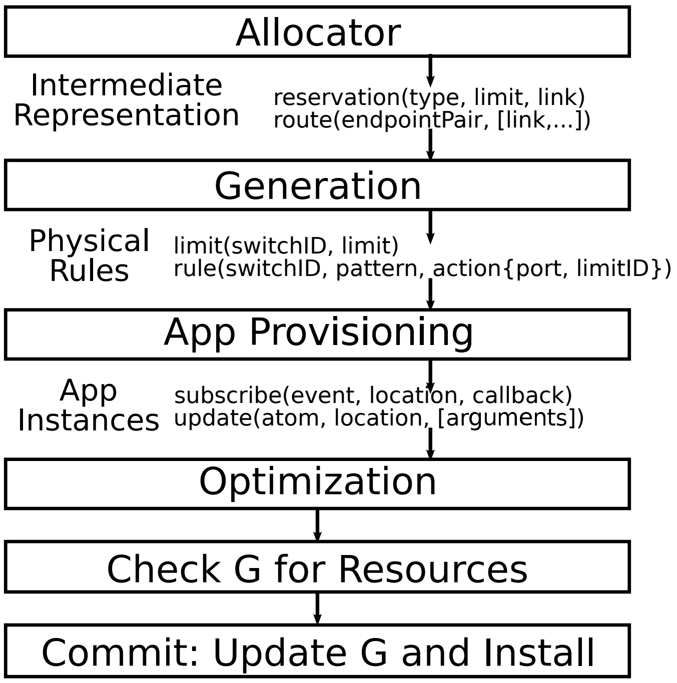

Allocator
Intermediate Representation
reservation(type, limit, link)
route(endpointPair, [link,...])
Generation
Physical Rules
limit(switchID, limit)
rule(switchID, pattern, action{port, limitID})
App Provisioning
App Instances
subscribe(event, location, callback)
update(atom, location, [arguments])
Optimization
Check G for Resources
Commit: Update G and Install

Figure 6.1. The Blender task compilation procedure over a graph, G, revised to include Omakase's App extensions. We describe Blender's other stages in Section 5.4.

Omakase inherits Blender's isolation models to provide performance isolation. We augment the isolation allocators to additionally designate zones as they annotate the graph representation of the network's resources. The performance isolation allocator is a natural location to assign zones, as isolation models commonly distinguish between multiple network areas to differentiate the resources they provision. For instance, we adapt the Squid allocator to divide its virtual networks into a zone for each of the clusters it allocates.

We implement App logic within reservation extension modules and introduce a new *App provisioning* stage to the task compilation process. The provisioning stage checks for (and when possible, resolves) static atom conflicts, instantiates App reservation extensions, and assigns the Apps' reservations to OpenFlow rules according to the zone specification in the tenant's service request. Figure 6.1 depicts our prototype implementation's architecture, updated to include Omakase's extensions. We place the

App provisioning stage before optimization to ensure that the optimizer takes reservation extensions into account (i.e., to prevent the optimizer from combining a set of rules for which a reservation only applies to a subset of the rules).

6.3.1 App execution

The Omakase model admits flexible placement of Apps on the network. Unlike other logically centralized systems, including Blender, functionality need not always execute at the controller. Depending on the App's requirements and the device capabilities, the App might execute at controller, on an end host (in the Blender shim), in a dedicated VM running App-specific software, on a middlebox, or directly in the network switches. In our Blender-based prototype, we modify Blender's shim to allow providers to include functionality in support of end-host atoms (e.g. notifications for new flows), and we label devices in the network graph with their atom capabilities, Apps choose where to execute under the assumption that sufficient processing resources are available. Other work [80] has explored the use of constraint satisfaction to schedule middlebox placement in resource-limited devices, and we believe similar techniques could be applied to the Omakase model to eliminate this assumption.

In addition to executing on end hosts and in the controller, we have tested placing Apps on an experimental Broadcom switch platform that supports low-level APIs for accessing the hardware. We ported Apps to execute on the switch's CPU, and Omakase correctly elected to execute on the switch, when possible. Unfortunately the device's TCAM update latency prevents us from collecting viable results in our evaluation of sample applications, so we limit our Apps to executing on conventional, general-purpose computing platforms.

Table 6.3. A summary of application classes and the atoms they require.

	Atom							
Application Class	**Create zones**	**Choose routes**	**Create rate limits**	**Modify rate limits**	**Read statistics**	**Assign flow paths**	**Inter-cept traffic**	**Modify traffic**
Performance isolation [8, 84, 101]	√	√	√	√				
Failure recovery		√	√					
Dynamic path selection [3, 42]					√	√		
Flow deadline scheduling [4, 100]				√	√			
Routing (external network) [78]		√					√	√
Load balancing [6, 98]					√	√		
VPN / Encryption [66]							√	√
Intrusion detection [88]							√	

6.4 Sample Apps

This section demonstrates Omakase's flexibility in implementing Apps. We describe several examples that are inspired from recent publications or open source projects and examine their interaction with Omakase. Table 6.3 lists the application classes we consider and summarizes the atoms they use. For each class, we examine a prototype implementation and evaluate its behavior.

6.4.1 Performance Isolation

Our Omakase prototype inherits its performance isolation functionality from Blender. As a result, we restrict performance isolation to be the only class of App that is allowed to define zones. We believe this is appropriate, since every tenant is likely to need some form of performance isolation. A detailed evaluation of Blender's performance isolation mechanisms and a description of our experimental testbed can be found in Chapter 5.

6.4.2 Failure Recovery

As the size of a network increases, so does the likelihood of a component failure. We recognize that different tenants may wish to respond to such failures in numerous ways, and the Omakase model allows for such diversity. For example, one tenant might

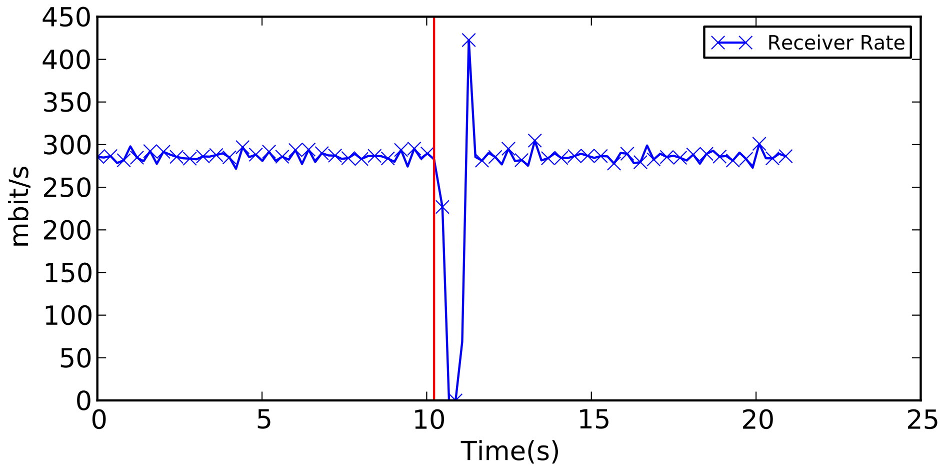


Figure 6.2. The receive rate of a traffic sink while running the failure recovery App. The vertical line at 10 seconds signals a switch failure event.

want to minimize the service interruption by automatically routing around the failure, a tenant second might want to reallocate a new virtual network in a different physical location, and a third may prefer to halt execution until the failure is resolved.

By default, Blender does not provide any mechanisms to automatically recover from link failures. For our Omakase prototype, we construct a failure recovery App that executes at the controller and operates like the first of the three examples described above. When a link failure occurs, the App determines which of a tenant's routes crossed that link and searches for an alternative path whose characteristics in the annotated network graph indicate that it could serve as an equivalent replacement. Upon finding such a link, it then applies the reservations from the failed link to the replacement and informs the appropriate end host shims to mark their future packets to use the replacement.

Figure 6.2 illustrates the failure recovery App's behavior. Within a pod of our fat-tree testbed network, we establish a path from a sending VM through the pod's aggregation layer to a receiving VM with a reservation of 300 Mbps. The sender uses the `iperf` tool to generate and send as much data as possible to the receiver via a TCP flow. We simulate a switch failure by configuring the aggregation switch to disconnect from the centralized controller. Upon detecting the switch's disappearance, the recovery App chooses to route around the failure using an alternative aggregation switch and establishes a 300-Mbps replacement reservation through it. In the figure, we plot the rate at which the receiver's NIC observes incoming data. We mark the time of the failure with a vertical line, after which the receiver stops seeing traffic for approximately 1.5 seconds. When the alternative path is established, the reception rate briefly spikes as the replacement switch's hardware rate policer accounts for the sudden burst of traffic, followed by a return to the expected 300-Mbps rate.

6.4.3 Flow Path Selection

When multiple routes connect a pair of communicating endpoints, Omakase selects only one route for any individual flow that passes between them to avoid TCP reordering problems. Here, we describe two Apps we have built to perform flow route selection. The first App, random assignment (RA), randomly maps each flow to an available route, which closely approximates conventional ECMP [42]. The second, bandwidth-maximizing assignment (BMA), is inspired by the global first-fit algorithm of Hedera [3]. BMA assigns a flow to the route with the largest available capacity.

Both flow assignment Apps are similarly structured, and we label each with the 'assign flow paths' atom. This label enables App subscription to a 'new flow' event notification, which is triggered by the end host hypervisor when a flow begins at one of its VMs. RA simply chooses a random path in response, whereas BMA additionally utilizes the 'read statistics' atom to read and compare path utilizations before returning a decision. While they currently perform placement for all new flows, either could easily be extended to selectively optimize only long-running or high-volume flows for bandwidth maximization.

To evaluate these Apps, we configure three paths between two VMs within a pod

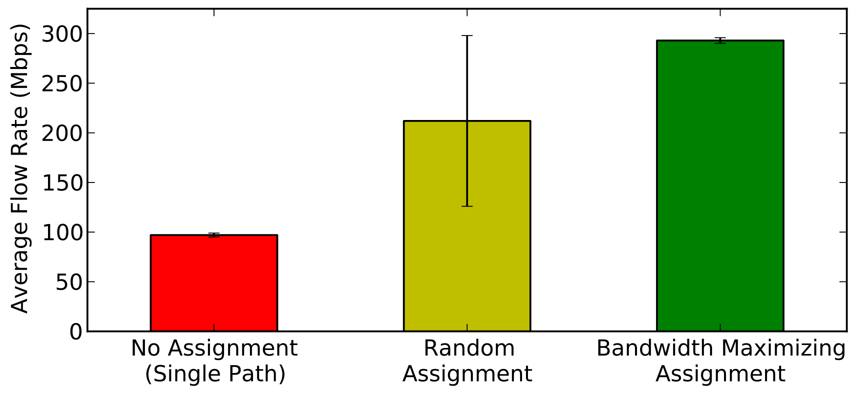


Figure 6.3. The average flow rate for three TCP flows, between two VMs, with three, 300-Mbps parallel paths. Error bars indicate standard deviation.

in our testbed. Each path holds a 300-Mbps reservation, and the sending VM transmits as much data as possible over three TCP flows using the `iperf` traffic generator. Figure 6.3 plots the average rate of the three flows over ten runs of the experiment, with error bars to designate the standard deviation.

6.4.4 Deadline-aware Flow Scheduling

While the conventional performance metric for data center applications has been bandwidth, several recent projects [4, 100, 106] have begun to focus on latency. This work is largely motivated by reports from industry [41], which indicate that even small increases in latency can lead to a significant reduction in customer satisfaction. For example, Amazon notes that a 100-ms increase in latency costs the company 1% in sales, and Google observes that an extra half second of search page generation time reduces customer traffic by 20%.

For the Omakase prototype, we constructed a deadline-aware App that was inspired by D3 [100], which uses deadlines to schedule flow completion times. To make use of the App, a tenant application informs its local Blender shim of the size and deadline for each new flow that it starts. The shim then begins periodically computing the rate necessary for the flow to complete on time and issues a request for that capacity to the App. At the same interval, the App considers all its received requests and produces a schedule of flow rates, which it uses to update switch rate policers.

Like D3, our App would ideally execute directly within the network's devices. Unfortunately, our experimental switch platform was not optimized for fast rate policer updates and could not keep up with the App's proposed changes. The poor performance is an issue of software (fast rate policer updating is not a frequently requested feature) and not a fundamental limitation of the hardware. To stand-in for programmable switches, we substitute the controller to execute the App, which reduces the responsiveness.

To demonstrate the App's behavior, we begin with a scenario that is similar to the previous experiment. Initially, a sending VM transmits via three TCP flows to a receiving VM across three parallel paths. Each path is configured with a 300-Mbps reservation. We use BMA flow placement to spread the flows across the available paths. This time, rather than sending as much data as possible, our traffic generator aims to send one gigabyte of data over each of the three flows with a deadline of 60 seconds. After 15 seconds, two additional flows begin, each with a size of 200 megabytes and a deadline of 10 seconds.

Figure 6.4 displays the results. Prior to the additional flows beginning, the three original flows converge to the necessary rate for on-time completion. When the two short-deadline flows start, our App displaces two of the longer flows until they complete. Afterwards, the two displaced flows converge on a higher rate to make up for their displaced time. The third flow, which never shared a link with one of the short-dealine flows, continues at its original rate.

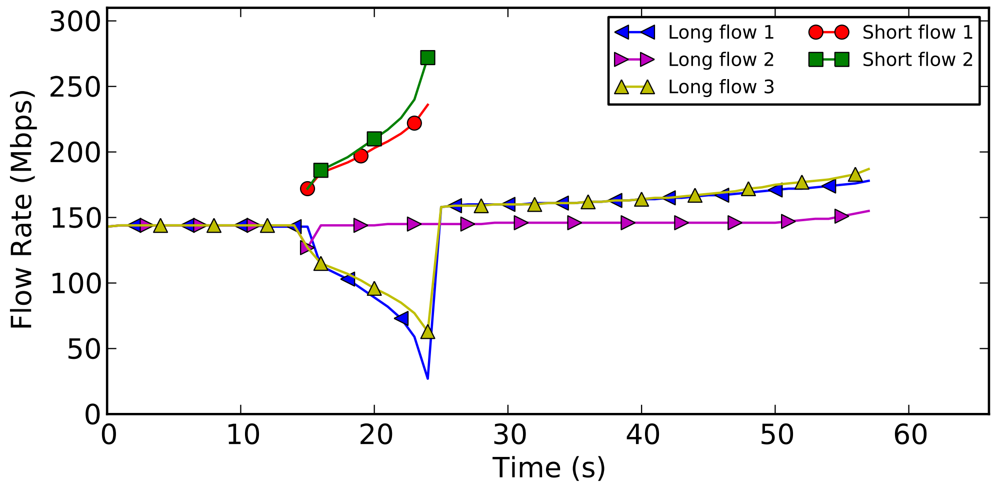


Figure 6.4. A combination of the BMA flow assignment App and a deadline-aware flow scheduling App. Long flows send one gigabyte of data with a deadline of 60 seconds, and short flows send 200 megabytes of data with a deadline of 10 seconds.

6.4.5 External Routing

To facilitate interaction with external (non-Blender) networks, we modified the BIRD [14] routing daemon to interface with the reservation extension infrastructure of our Omakase prototype. The resulting routing App's functionality resembles the RouteFlow [78] project's goal of integrating traditional routing protocols into a software-defined networking control architectures. However, rather than using a dedicated VM to host the routing daemon, we adjust the (comparatively lightweight) BIRD daemon process to establish a standard socket connection with the Blender controller.

The routing App leverages the 'intercept traffic' atom to place rules in switches that capture routing protocol messages and forward them to the controller. The controller then relays the routing messages via a socket connection to the BIRD instance responsible for the switch that received the message. The controller marks each relayed protocol message with the switch port it was received on, and our modifications to BIRD cause it to perceive reception of the message on a corresponding virtual network interface. We similarly capture the resulting decisions made by the BIRD daemon (i.e., routing table entry changes) and relay them to the routing App, which pushes the equivalent OpenFlow rules into the switch. With this relay architecture, the routing daemon may run on the same server as the controller, or it could be offloaded to a secondary control machine if the controller's processing burden becomes a concern.

6.4.6 Middlebox Apps

Middleboxes represent a popular method for interjecting new functionality into networks [82]. To demonstrate that Omakase can effectively enable middleboxes, we introduced our prototype implementation to several small groups of graduate students. Over the course of a five-week graduate networking class, they implemented various forms of middlebox Apps as a class project. In each of their projects, their App used at least one dedicated VM to provide their middlebox functionality. The VM ran open source middlebox software (e.g., Snort [88]) and traffic directly passed through the middlebox without inspection by the controller. The range of Apps they produced included an intrusion detection system [60], a data compression service [45], a virtual private networking App [108], and a wide-area network optimization utility [81].

6.5 Summary

Researchers, open source communities, and network hardware vendors have undertaken many useful and interesting software ventures to improve cloud networks. Unfortunately, individual projects typically solve only a small number of the problems faced by cloud tenants, and current cloud computing services limit the ability of tenants to receive the complete set of functionality they want from the network. To complicate matters, in highly-shared cloud environments, operational requirements are likely to vary considerably from one tenant to another.

In this chapter, we presented Omakase, a model for building modular Apps for the data center network. Omakase allows service operators to easily pick and choose the pieces of functionality that best meet their needs. We demonstrated Omakase's main contributions:

- **Omakase tenant model:** Omakase defines a cloud networking model in which network providers can easily author new Apps to enhance the network with new functionality for tenants. Apps declare their use of atoms, which unlocks event subscriptions and network state modifications. Omakase admits App execution in multiple locations within in network and distinguishes between zones to more easily validating the coexistence of a set of Apps.

- **Implementation and evaluation:** We illustrate the effectiveness of the Omakase model with a prototype implementation that extends the Blender model. We show that Omakase's extensions are flexible and capable of naturally augmenting the network with valuable features including failure recovery, flow path assignment, and deadline-aware flow scheduling. Furthermore, we demonstrate that the model is straightforward to use as evidenced by several groups of students who were able to acquaint themselves with the codebase and construct middlebox Apps in a short time frame as part of a UCSD networking course.

In the following chapter, we conclude the dissertation and discuss avenues for future research.

Acknowledgements

Chapter 6 contains material that is currently being prepared for submission. “An App-store Framework for Data Center Networks”. Webb, Kevin; Roy, Arjun; Yocum, Ken; Snoeren, Alex. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusion

Recent technical and economic trends have fueled the construction of large and sophisticated data centers. Containing thousands of servers, their size and stringent performance requirements represent a unique operating environment with critical challenges for their interconnection networks. Conventional topologies and routing systems cannot cope with the massive level of redundancy needed to achieve fault tolerance and provide high performance. Furthermore, these environments host a diverse set of services executing side-by-side, each with potentially contrasting operating parameters, contributing to an overwhelming complexity of resource allocation and device configuration.

In spite of these challenges, data centers also represent a rare opportunity to leverage the exclusive ownership of a large installation to take advantage of service diversity and deliver qualitatively different paths to tenant applications. In this chapter, we summarize our vision, architecture, and prototype implementation for providing a custom, virtualized network for every tenant and discuss the current limitations and potential future directions of the work described here.

7.1 Summary

With task switching, we presented a new architecture for data center services to interact with the network. In contrast to other proposals that allocate resources to flows or VMs, task switching allocates resources at the unit of a *task*, which encompasses all of the resources associated with a tenant's service. Towards the goal of providing modularity for network providers and flexibility for tenants, task switching rejects the one-size-fits-all approach to network path selection to provide services with the ability to customize their routing and resource allocation decisions.

In a task-switched network, a centralized topology server maintains a view of the network's structure and available resources. Tenants submit requests for resources to the topology server, which uses one of multiple allocators to assign resources to the tenant according to the content of the request and the network's available capacity. We described and simulated one possible set of allocators that allow tenant services to take advantage of paths that others choose not to utilize, and our simulations demonstrated the allocators' effectiveness for symmetric and heterogeneous task workloads. Task switching serves as the foundation for our Blender performance isolation model and its Omakase application extensions.

The Blender model represents a realization of the task switching framework with an emphasis on inter-task performance isolation. It resolves the either/or balancing act between resource efficiency and performance predictability by allowing data center operators to run multiple performance isolation models simultaneously. The model simplifies the ability of data center providers to compose different isolation models by allocating over a hierarchical set of logical network graphs. Blender builds on many recently-proposed isolation models as well as on advancements for controlling software-defined networks.

Blender can express a variety of task-based isolation models that provide both fixed and weighted capacity sharing across data center network links. Put within the context of our task switching architecture, it becomes possible for data center operators to easily author and compose multiple isolation models across a shared physical infrastructure. Moreover, our experiments with a prototype implementation indicate that task-based isolation models can be practical given the current rate limiter facilities in modern top-of-rack switches.

With Omakase, we introduced a model for augmenting task switching with general App functionality that goes beyond forwarding and capacity guarantees. Omakase allows cloud-computing tenants to succinctly express the functionality they want from their cloud network in a fashion that resembles users extending their baseline smartphone functionality via user-selected apps. Tenants can choose and deploy software to control the reliability, routing, fairness, security, and latency of their virtual network infrastructure with the same ease and flexibility with which they configure virtualized compute nodes.

Our Omakase prototype extends Blender with network zones and novel conflict resolution routines to detect and resolve situations in which multiple Apps conflict at shared locations in the network. Apps utilize straightforward APIs to declare their use of atoms, which unlock event subscriptions and network state modification opportunities. Additionally, we demonstrate the effectiveness of the Omakase model by characterizing several example Apps and describe how they could take advantage of in-network processing capabilities to go beyond standard software-defined networking platforms.

7.2 Limitations and Future Directions

The systems and analysis described in this dissertation are not without limitations. For example, while technical solutions are necessary to achieve success in providing virtual network services, an equally critical barrier to adoption stems from economics. How much more should Blender's guaranteed rate reservation cost than a proportional share? How expensive should adding flow assignment or flow deadline functionality be to a tenant? In this dissertation, we made no attempt to quantify the financial expense of our proposed mechanisms or prescribe potential tenant costs. In a practical deployment, the data center provider must know the operational cost to reserve capacity or execute an App to ensure that tenants are paying appropriately for network resources. Due to the competitive and variable nature of cloud computing markets, we believe an economic analysis of flexible data center resource resource assignment is an important avenue for complementary future work.

A second obstacle that complicates the deployment of our task switching systems is that current network hardware platforms have not been optimized for *fast* programmability and reconfiguration. SDN and OpenFlow have made great strides towards an open and programmable device model, but much of the currently available, SDN-capable hardware was retrofitted rather than designed for SDN from scratch. Such devices have been engineered for the common case of infrequent routing or policy changes and thus they respond slowly to data path updates, even when the underlying hardware (e.g., the TCAM flow table) is theoretically capable of much higher performance. The result is that installing new rules for large tasks or executing high frequency feedback loops may not be feasible on current-generation commodity hardware.

Another consequence of limited hardware support for SDN is that, despite conforming to a standardized API, current SDN devices commonly ignore useful directives, relegate nonessential functionality to slow software paths, or place additional restrictions on hardware usage (e.g., only a subset of the packet header can be matched). Ultimately, we hope to see switch hardware and software co-designed from the ground up for high-speed programmability.

A co-designed platform would convey several benefits to task-switched systems. Ideally, such a platform would contain a strong CPU (to balance the device's general-purpose processing performance with the high-speed forwarding hardware), an API for bulk rule installation (making it easier to efficiently provision and optimize large tasks), and better accessibly for executing code on the device (allowing for devices to trigger events rather than relying on less-efficient controller polling). Fortunately, network device vendors are beginning to design and position their hardware in this way, and it is now only a matter of time until commodity devices start to match this vision.

Finally, with 50 servers, we have reached the limits to the size of an academic testbed, and the prototypes presented here would benefit from further evaluation at a larger scale. Further, while we have attempted to measure our prototypes under reasonable assumptions, the research community desperately needs additional data and guidance from industry regarding tenants, workload, and network traffic characteristics. Network virtualization is an important problem in industry, and establishing a robust solution is a necessary step for the cloud computing model to meet its full potential. Several commercial systems are currently being developed, and though the final product may not look like Blender, we believe the central ideas introduced in this dissertation have strong advantages over other approaches and could be integrated into a commercial network virtualization solution.

Bibliography

[1] Jung Ho Ahn, Nathan Binkert, Al Davis, Moray McLaren, and Robert S. Schreiber. HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks. In *SC09*, 2009.

[2] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[3] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *NSDI*, 2010.

[4] Mohammad Alizadeh, Abdul Kabbani, Tom Edsall, Balaji Prabhakar, Amin Vahdat, and Masato Yasuda. Less is More: Trading a little Bandwidth for Ultra-Low Latency in the Data Center. In *NSDI*, 2012.

[5] Amazon. Elastic Compute Cloud (EC2). http://aws.amazon.com/ec2.

[6] Amazon. Elastic Load Balancing. http://aws.amazon.com/elasticloadbalancing.

[7] Andrew Bach. Higher- and faster-than-ever trading to transform data center networks. Datacenter Dynamics, 2009.

[8] Hitesh Ballani, Paolo Costa, Thomas Karagiannis, and Ant Rowstron. Towards Predictable Datacenter Networks. In *SIGCOMM*, 2011.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *SOSP*, 2003. Current open source project: http://www.xen.org.

[10] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM*, 2006.

[11] Theophilus Benson and Aditya Akella. CloudNaaS: A Cloud Networking Platform for Enterprise Applications. In *SOCC*, 2011.

[12] Theophilus Benson, Aditya Akella, and David A. Maltz. Network Trafic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[13] Theophilus Benson, Ashok Anand, Aditya Akella, and Ming Zhang. The Case for Fine-Grained Traffic Engineering in Data Centers. In *WREN*, 2010.

[14] BIRD. The BIRD Internet Routing Daemon. http://bird.network.cz.

[15] Martin Casado, Michael Freedman, Justin Pettit, Jianying Luo, Nick McKeown, and Scott Shenker. Ethane: Taking Control of the Enterprise. In *SIGCOMM*, 2007.

[16] Cisco. *Data Center: Load Balancing Data Center Services, Solutions Reference Network Design*, March 2004.

[17] Cisco. *Quality of Service on the Cisco Catalyst 4500 Classic Supervisor Engines*, 2006.

[18] Cisco. *Scaling Data Centers with FabricPath and the Cisco FabricPath Switching System*, 2010.

[19] Paolo Costa, Matteo Migliavacca, Peter Pietzuch, and Alexander L. Wolf. NaaS: Network-as-a-Service in the Cloud. In *Hot-ICE*, 2012.

[20] Andrew R. Curtis, Wonho Kim, and Praveen Yalagandula. Mahout: Low-Overhead Datacenter Traffic Management using End-Host-Based Elephant Detection. In *INFOCOM*, 2011.

[21] Andrew R. Curtis, Jeffrey C. Mogul, Jean Tourrilhes, Praveen Yalagandula, Puneet Sharma, and Sujata Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[22] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[23] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Innaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *SOSP*, 2009.

[24] David Erickson. Beacon. https://openflow.stanford.edu/display/Beacon/Home.

[25] Facebook. Press Release: Facebook Opens First Data Center in Prineville, Oregon. April 15, 2011.

[26] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *SIGCOMM*, 2010.

[27] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004.

[28] Nate Foster, Rob Harrison, Michael J. Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A Network Programming Language. In *ICFP*, 2011.

[29] The Apache Software Foundation. Hadoop. http://hadoop.apache.org.

[30] David Gauthier. Microsoft Cloud-Scale Data Center Designs. Microsoft Data Centers Blog, March 2013. http://www.globalfoundationservices.com/posts/2013/march/26/microsoft-cloud-scale-data-center-designs.aspx.

[31] Google. Data Center Locations. http://www.google.com/about/datacenters/inside/locations.

[32] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: A scalable and Flexible Data Center Network. In *SIGCOMM*, 2009.

[33] 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 CCR*, 35(5), October 2005.

[34] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker. NOX: Towards an Operating System for Networks. *SIGCOMM CCR*, 2008.

[35] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. Bcube: A high performance, server-centric network architecture for modular data centers. *SIGCOMM*, 2009.

[36] Chauanxiong Guo, Guohan Lu, Helen Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Zhang Yongguang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *CoNEXT*, 2010.

[37] Chuanxiong Guo, Haiao Wu, Kun Tan, Lei Shi, Yongguang Zhang, and Songwu Lu. DCell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, 2008.

[38] Daniel Halperin, Srikanth Kandula, Jitendra Padhye, Paramvir Bahl, and David Wetherall. Augmenting Data Center Networks with Multi-Gigabit Wireless Links. In *SIGCOMM*, 2011.

[39] James Hamilton. Data center networks are in my way. Talk: Stanford Clean Slate CTO Summit, 2009.

[40] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *SIGCOMM*, 2010.

[41] Todd Hoff. Latency is Everywhere and it Costs You Sales, July 2009. http://highscalability.com/blog/2009/7/25/latency-is-everywhere-and-it-costs-you-sales-how-to-crush-it.html.

[42] C. Hopps. Analysis of an Equal-Cost Multi-Path Algorithm. IETF RFC 2992, 2000.

[43] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks. In *Eurosys*, 2007.

[44] Sushant Jain, Alok Kumar, Subhasree Mandal, Joon Ong, Leon Poutievski, Arjun Singh, Subbaiah Venkata, Jim Wanderer, Junlan Zhou, Min Zhu, Jonathan Zolla, Urs Holzle, Stephen Stuart, and Amin Vahdat. B4: Experience with a Globally-Deployed Software Defined WAN. In *SIGCOMM*, 2013.

[45] Matthew Johnson, Sandy Law, and Constantine Chen. Data Compression on Blender. In *Course project for CSE222A: Computer Communication Networks (UCSD)*, Winter 2013.

[46] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[47] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichior Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[48] KVM. Kernel Based Virtual Machine. http://www.linux-kvm.org/page/Main_Page.

[49] Vinh The Lam, Sivasankar Radhakrishnan, Amin Vahdat, George Varghese, and Rong Pan. NetShare and Stochastic NetShare: Predictable Bandwidth Allocation for Data Centers. *SIGCOMM CCR*, 42(3), 2012.

[50] Lydia Leong and Ted Chamberlin. Magic Quadrant for Public Cloud Infrastructure as a Service. Gartner, December 2011.

[51] Dionysios Logothetis and Kenneth Yocum. Wide-scale data stream management. In *USENIX*, 2008.

[52] Guohan Lu, Chuanxiong Guo, Yulong Li, Zhiqiang Zhou, Tong Yuan, Haitao Wu, Yongqiang Xiong, Rui Gao, and Yongguang Zhang. ServerSwitch: A Programmable and High Performance Platform for Data Center networks. In *NSDI*, 2011.

[53] Navneet Malpani and Jianer Chen. A note on practical construction of maximum bandwidth paths. *Inf. Process. Lett.*, 83:175–180, August 2002.

[54] Tal Garfinkel Martin Casado, Aditya Akella, Michael J. Freedman, Dan Boneh, Nick McKeown, and Scott Shenker. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security*, 2006.

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

[56] X. Meng, V. Pappas, and L. Zhang. Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement. In *INFOCOM*, 2010.

[57] Microsoft. Windows Azure. http://www.windowsazure.com.

[58] Rich Miller. Inside Microsoft's Chicago Data Center. Data Center Knowledge, October 2009. http://www.datacenterknowledge.com/inside-microsofts-chicago-data-center.

[59] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, and David Walker. Composing Software-Defined Networks. In *NSDI*, 2013.

[60] James Mouradian, Devin Lundberg, and John Chau. Pigs in a Blender: Running Snort NIDS on a Software Dened Network. In *Course project for CSE222A: Computer Communication Networks (UCSD)*, Winter 2013.

[61] Jayaram Mudigonda, Praveen Yalagandula, Mohammad Al-fares, and Jeffrey C Mogul. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *NSDI*, 2010.

[62] Jayram Mudigonda, Praveen Yalagandula, Bryan Stiekes, Jeffrey Mogul, and Y. Pouffary. Netlord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.

[63] 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*, 2009.

[64] Institute of Electrical and Electronics Engineers. 802.1aq Shortest Path Bridging, 2010. http://www.ieee802.org/1/pages/802.1aq.html.

[65] Office of the Governor of Iowa, Terry Branstad. Facebook Chooses Iowa for Next Data Center Location. April 23, 2013.

[66] OpenVPN. http://openvpn.net.

[67] Oracle. Oracle VM Server for x86. http://www.oracle.com/us/technologies/virtualization/oraclevm/overview/index.html.

[68] Kostas Pagiamtzis and Ali Sheikholeslami. Content-Addressable Memory (CAM) Circuits and Architectures: A Tutorial and Survey. *IEEE Journal of Solid-State Circuits*, 41(3), March 2006.

[69] Craig Partridge, Alex C. Snoeren, W. Timothy Strayer, Beverly Schwartz, Matthew Condell, and Isidro Castineyra. FIRE: Flexible Intra-AS Routing Environment. In *SIGCOMM*, 2000.

[70] Pankesh Patel, Ajith Ranabahu, and Amit Sheth. Service Level Agreement in Cloud Computing. In *Cloud Workshops at OOPSLA*, 2009.

[71] Larry Peterson, Andy Bavier, Marc E. Fiuczynski, and Steve Muir. Experiences Building PlanetLab. In *OSDI*, 2006.

[72] Lucian Popa, Gautam Kumar, Mosharaf Chowdhury, Arvind Krishnamurthy, Sylvia Ratnasamy, and Ion Stoica. FairCloud: Sharing the Network in Cloud Computing. In *SIGCOMM*, 2012.

[73] Project Floodlight. Floodlight. http://www.projectfloodlight.org/floodlight.

[74] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud Control with Distributed Rate Limiting. In *SIGCOMM*, 2007.

[75] Robert Ricci, Chris Alfeld, and Jay Lepreau. A solver for the network testbed mapping problem. *ACM SIGCOMM CCR*, 32(2), 2003.

[76] Henrique Rodrigues, Jose Renato Santos, Yoshio Turner, Paolo Soares, and Dorgival Guedes. Gatekeeper: Supporting Bandwidth Guarantees for Multi-tenant Datacenter Networks. In *WIOV*, 2011.

[77] Sean Rooney, Jacobus E. van der Merwe, Simon A. Crosby, and Ian M. Leslie. The Tempest: A Framework for Safe, Resource-Assured, Programmable Networks. *IEEE Communications*, October 1998.

[78] Christian E. Rothenberg, Marcelo R. Nascimento, Marcos R. Salvador, Carlos N. A. Correa, Sidney C. de Lucena, and Robert Raszuk. Revisiting routing control platforms with the eyes and muscles of software-defined networking. In *HotSDN*, 2012.

[79] SDN Central. Shipping SDN Products. http://www.sdncentral.com/shipping-sdn-products.

[80] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K. Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *NSDI*, 2012.

[81] Yichao Shen, Apurva Kumar, and Boxiang Pan. Adding WAN optimizing functionality on Blender. In *Course project for CSE222A: Computer Communication Networks (UCSD)*, Winter 2013.

[82] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making Middleboxes Someone Else's Problem: Network Processing as a Cloud Service. In *SIGCOMM*, 2012.

[83] Rob Sherwood, Glen Gibb, Kok-Kiong Yap, Guido Appenzeller, Martin Casado, Nick McKeown, and Guru Parulkar. Can the Production Network Be the Testbed? In *OSDI*, 2010.

[84] Alan Shieh, Srikanth Kandula, Albert Greenberg, Changhoon Kim, and Bikas Saha. Sharing the Data Center Network. In *NSDI*, 2011.

[85] Alan Shieh, Srikanth Kandula, and Emin Gun Sirer. SideCar: Building Programmable Datacenter Networks without Programmable Switches. In *HotNets*, 2010.

[86] Seugwon Shin, Phillip Porras, Vinod Yegneswaran, Martin Fong, Guofei Gu, and Mabry Tyson. FRESCO: Modular Composable Security Services for Software-Defined Networks. In *NDSS*, 2013.

[87] Ankit Singla, Chi-Yao Hong, Lucian Popa, and P. Brighten Godfrey. Jellyfish: Networking Data Centers Randomly. In *NSDI*, 2012.

[88] Snort. http://www.snort.org.

[89] Tammo Spalink, Scott Karlin, Larry Peterson, and Yitzchak Gottlieb. Building a Robust Software-Based Routing Using Network Processors. In *SOSP*, 2001.

[90] Herb Sutter. The Free Lunch is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3), March 2005.

[91] J. Touch and R. Perlman. RFC 5556: Transparent interconnection of lots of links (TRILL), May 2009.

[92] James Urquhart. Amazon APIs as Cloud Standards? Not so Fast. CNET Commentary, July 2010. http://news.cnet.com/8301-19413_3-20010072-240.html.

[93] VMWare. vSphere ESX and ESXi. http://www.vmware.com/products/vsphere/esxi-and-esx/overview.html.

[94] Andreas Voellmy, Ashish Agarwal, and Paul Hudak. Nettle: Functional Reactive Programming for OpenFlow Networks. Technical Report YALEU/DCS/RR-1431, Yale University, July 2010.

[95] Stephen R. Walli. The POSIX Family of Standards. *StandardView*, 3(1), March 1995.

[96] Gouhui Wang and T.S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *INFOCOM*, 2010.

[97] Guohui Wang, David G. Andersen, Michael Kaminsky, Konstantina Papagiannaki, T. S. Eugene Ng, Michael Kozuch, and Michael Ryan. c-Through: Part-time Optics in Data Centers. In *SIGCOMM*, 2010.

[98] Richard Wang, Dana Butnariu, and Jennifer Rexford. OpenFlow-Based Server Load Balancing Gone Wild. In *Hot-ICE*, 2011.

[99] David Wetherall. Active Network Vision and Reality: Lessons from a Capsule-Based System. In *SOSP*, 1999.

[100] Christo Wilson, Hitesh Ballani, Thomas Karagiannis, and Ant Rowstron. Better Never than Late: Meeting Deadlines in Datacenter Networks. In *SIGCOMM*, 2011.

[101] Di Xie, Ning Ding, Y. Charlie Hu, and Ramana Kompella. The Only Constant is Change: Incorporating Time-Varying Network Reservations in Data Centers. In *SIGCOMM*, 2012.

[102] Hong Yan, David A. Maltz, T.S. Eugene Ng, Hermant Gogineni, Hui Zhang, and Zheng Cai. Tesseract: A 4D Network Control Plane. In *NSDI*, 2007.

[103] Minlan Yo, Jennifer Rexford, Michael J. Freedman, and Jia Wang. Scalable Flow-Based Networking with DIFANE. In *SIGCOMM*, 2010.

[104] Minlan Yu, Yung Yi, Jennifer Rexford, and Mung Chiang. Rethinking virtual network embedding: Substrate support for path splitting and migration. *ACM SIGCOMM CCR*, 38(2), April 2008.

[105] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Envrionments. In *OSDI*, 2008.

[106] David Zats, Tathagata Das, Prashanth Mohan, Dhruba Borthakur, and Randy Katz. DeTail: Reducing the Flow Completion Time Tail in Datacenter Networks. In *SIGCOMM*, 2012.

[107] Alexander Z. Zelikovsky. A Faster Approximation Algorithm for the Steiner Tree Problem in Graphs. *Information Processing Letters*, 46(2), May 1993.

[108] Xiang Zhou, Wenwen Zhou, and Xiaofen Wei. VPN Implementation on Blender. In *Course project for CSE222A: Computer Communication Networks (UCSD)*, Winter 2013.