Technische Universität Berlin
Fakultät für Elektrotechnik und Informatik
Lehrstuhl für Intelligente Netze
und Management Verteilter Systeme

Towards Improved Control and Troubleshooting for Operational Networks

vorgelegt von
Andreas Wundsam (Dipl.-Inf.)

von der Fakultät IV – Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften (Dr.-Ing.)
genehmigte Dissertation

Promotionsausschuss:

Vorsitzender:	Prof. Dr. Jean-Pierre Seifert, TU Berlin
Gutachterin:	Prof. Anja Feldmann, Ph. D., TU Berlin
Gutachter:	Prof. Dr. Laurent Mathy, Lancaster University, UK
Gutachter:	Dr. Olaf Maennel, Loughborough University, UK

Tag der wissenschaftlichen Aussprache: 15. Juli 2011

Berlin 2011
D83

Ich versichere an Eides statt, dass ich diese Dissertation selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

Datum Andreas Wundsam

Abstract

Over the past decade, operational networks, have grown tremendously in size, performance and importance. This concerns particularly the Internet, the ultimate "network of networks." We expect this trend to continue as more and more services traditionally provided by the local computer move to the *cloud*, e.g., file storage services and office applications.

In spite of this, our ability to control and manage these networks remains painfully inadequate, and our visibility into the network limited. This has been exemplified by several recent outages that have caused significant disruption of important Internet services [24, 14, 149, 126].

Part of the challenges for controlling and troubleshooting networks stem from the nature of the problem: Networks are intrinsically highly distributed systems with distributed state and configuration. Consequently, a consistent view of the network state is often difficult to attain. They are also highly heterogeneous: Their scale ranges from small home-networks to data center networks that transfer enormous amounts of data at high speeds between thousands of hosts. Their geographic spread may be confined to a single rack, or span the globe. The Internet combines all these different kinds of networks, and thus their individual challenges.

In addition, the network architecture and the available toolset has evolved little if at all over the past decade. In fact, the Internet core and architecture has been diagnosed with *ossification* [48]. Thus, debugging problems in an operational network still comes down to guesswork, as the architecture provides little support for fault localization and troubleshooting, and available tools like `NetFlow`, `traceroute` and `tcpdump` provide either only coarse-grained statistical insight, or are confined to single vantage points and do not provide consistent information across the network.

In this thesis, we explore how to improve our control over networks and our abilities to debug and troubleshoot problems. Due to the extreme diversity of the environments, we do not strive for a one-size-fits-all solution, but propose and evaluate several approaches tailored to specific important scenarios and environments. We emphasize *network centric* approaches that can be implemented locally and are transparent to the end hosts. In the spirit of trusting "running code", we implement all our approaches "on the metal" and evaluate them in real networks.

We first explore the *Potential of Flow Routing* as an approach available to end users to self-improve their Internet Access. We find Flow-Routing to be a viable, cost-efficient approach for communities to share and bundle their access lines for improved reliability and performance.

On a wider scale, we explore Network Virtualization as a possible means to to overcome the ossification of the Internet core and also enable new troubleshooting primitives. We propose a *Control Architecture for Network Virtualization* in a multi-player, multi-role scenario.

We next turn to troubleshooting. Based on Network Virtualization, we propose *Mirror VNets* as a primitive that enables safer evolution and improved debugging abilities for complex network services. To this end, a production VNet is paired with a *Mirror VNet* in identical state and configuration.

Finally, we explore how Software Defined Network architectures, e.g., OpenFlow, can be leveraged to enable record and replay troubleshooting for Networks. We propose and evaluate *OFRewind*, the first system that enables practical record and replay in operational networks, even in the presence of *black-box devices* than cannot be modified or instrumented. We present several case studies that underline its utility. Our evaluation shows that OFRewind scales at least as well as current controller implementations and does not significantly impact the scalability of an OpenFlow controller domain.

In summary, we propose several simple but effective, scenario-specific and network centric approaches that improve the control and troubleshooting of Operational Networks, from the residential network and access line to the datacenter. Our approaches have all been implemented and evaluated on real networks, and can serve as a datapoint and guidance for how networks may need to evolve to cater to their growing importance.

Zusammenfassung

Während des letzten Jahrzehnts haben Netzwerke, und besonders das Internet als "Netz der Netze", in hohem Maße an Bedeutung gewonnen. Gleichzeitig ist auch ihre Geschwindigkeit und ihre Ausdehnung stark gewachsen. Dieser Trend wird sich absehbar fortsetzen: Heute bereits wandern mehr und mehr Dienste vom lokalen PC in die "Cloud", zum Beispiel Daten-Sicherungen, aber auch Office-Applikationen. Dadurch wird die Zuverlässigkeit der Netze für unser tägliches Leben immer wichtiger.

Trotz alledem sind bis heute unsere Möglichkeiten, diese Netze sicher zu verwalten, und Fehler zu beseitigen und zu beheben, stark beschränkt und reichen nicht aus. Häufig haben wir nur eingeschränkten Einblick in das, was in den Netzen passiert. In letzter Zeit gab es mehrere aufsehenerregende Ausfälle von wichtigen Internet-Diensten, die das deutlich gemacht haben [24, 14, 149, 126].

Einige der Gründe für die Schwierigkeiten, Netze sicher zu verwalten und Fehler zu finden, liegen in der Natur der Angelegenheit: Netzwerke sind inhärent hoch komplexe verteilte Systeme, und ihr Zustand und Konfiguration verteilen sich auf viele Einzelknoten. Deshalb ist es oft schwierig, einen konsistenten Überblick über ihren Zustand zu gewinnen. Sie sind auch in hohem Maße heterogen: Ihre Größe rangiert von kleinen, leeren Heim-Netzwerken bis zu Netzwerken in Data-Centern, die enorme Datenmengen zwischen zehntausenden Rechnern austauschen. Ihre geographische Ausdehnung kann sich auf einen einzelnen Serverschrank oder auf mehre Kontinente erstrecken. Das Internet vereint all diese unterschiedlichen Netzwerke und damit auch deren Herausforderungen.

Zusätzlich haben sich weder die Architektur unserer Netze noch unsere Werkzeuge in den letzten Jahren angemessen weiterentwickelt. Deshalb wurde der Internet-Architektur und dem Internet-Core in den vergangenen Jahren "Verknöcherung" attestiert [48]. Dies hat zur Folge, dass Fehler in echten Netzen auch heute noch oft nur durch Ausprobieren und Raten gefunden und behoben werden können, weil die Internet-Architektur nur wenige Mechanismen zur Fehlersuche bereitstellt, und Werkzeuge wie *NetFlow*, *traceroute* und *tcpdump* entweder nur grobkörnige statistische Informationen liefern, oder auf einen einzigen Beobachtungspunkt beschränkt sind, und kein konsistentes Bild des Netzwerkes liefern können.

In dieser Dissertation untersuche ich, wie die Kontrolle über unsere Netze und unsere Fähigkeit zur Problemfindung und -behebung verbessert werden kann. Wegen der großen Bandbreite der unterschiedlichen Umgebungen suche ich dabei nicht nach einer alles umfassenden Einheitslösung. Statt dessen schlage ich mehrere Ansätze vor, die auf spezifische, relevante Szenarien und Umgebungen zugeschnitten sind. Ich konzentriere mich auf *netzwerk-zentrische* Lösungen, die lokal implementiert werden können und für die Endgeräte transparent sind. Im Sinne des Internet-Credos, nur "laufendem Code" zu vertrauen, wurden die untersuchten Ansätze "auf dem Blech" implementiert und in echten Netzen evaluiert.

Zuerst untersuche ich das *Potential von Flow-Routing*, einem Ansatz, mit dem End-Benutzer die Zuverlässigkeit und Geschwindigkeit ihres Internet-Anschlusses selbst verbessern können. Die Ergebnisse zeigen, dass Flow-Routing eine sinnvolle, kosteneffiziente Möglichkeit sein kann, Internet-Anschlüsse in Gruppen zu teilen und zu verbinden, und damit Zuverlässigkeit und Geschwindigkeit zu verbessern.

Im größeren Maßstab untersuche ich dann Netzwerkvirtualisierung als Möglichkeit, die "Verknöcherung" des Internet-Kerns zu beheben und neue Möglichkeiten für die Fehlerbehebung und Analyse zu schaffen. Ich schlage eine *Kontroll-Architektur für Virtuelle Netze* vor, die auf eine Umgebung mit mehreren konkurrierenden Akteuren zugeschnitten ist.

Danach widme ich mich konkret der Fehlerbehebung. Aufbauend auf Virtuellen Netzen schlage ich *Mirror VNets* vor, die eine sichere Fortentwicklung und Online-Fehlersuche und -behebung für komplexe Netzwerkdienste ermöglichen. Dazu wird ein Produktions-VNet mit einem "Spiegelnetz" kombiniert, das in identischer Zustand und Konfiguration erzeugt wird. Die Fehlersuche, das Upgrade oder die Rekonfiguration kann dann sicher im Spiegelnetz erfolgen, erst im Erfolgsfall werden die Netze umgeschaltet.

Zuletzt wende ich mich der Server-Seite des Internet zu. Ich untersuche, wie neuartige Architekturen für "Software Defined Networks", wie z.B. OpenFlow, uns helfen können, Fehler in Netzwerken schneller zu finden und zu beheben. Ich schlage *OFRewind* vor, das erste System, das es ermöglicht, Netze aufzunehmen und wieder abzuspielen – das gelingt sogar dann, wenn diese Netze geschlossene "Black-boxen" enthalten, z.B. kommerzielle Router und Switches, die nicht verändert oder instrumentiert werden können. Ich präsentiere mehrere Fallstudien, die die Anwendbarkeit von OFRewind zeigen. Außerdem untersuche ich seine Skalierbarkeit und zeige, dass es mindestens so gut wie aktuell übliche Controller-Implmentierungen skaliert, und deshalb die Skalierbarkeit eines OpenFlow-Netzes nicht signifikant beeinflusst.

Zusammengefasst schlage ich mehrere einfache, aber effiziente, szenario-spezifische und netzwerk-zentrische Ansätze vor, die die Kontrolle und Fehlerbehebung für Netzwerke verbessern können, vom Heimnetz über die hemische Internet-Leitung bis zum großen Datacenter. Alle Ansätze wurden praktisch implementiert und in echten Netzen evaluiert. Sie können daher als Hinweisgeber dafür dienen, wie Netzwerke sich weiterentwickeln müssen, um ihrer wachsenden Bedeutung für unseren Alltag gerecht werden zu können.

Pre-published Papers

Parts of this thesis are based on pre-published papers co-authored with other researchers. I thank all of my co-authors for their valuable contributions! All co-authors have been acknowledged as scientific collaborators of this work.

Wundsam, A., Levin, D., Seetharaman, S., and Feldmann, A. **OFRewind: Enabling Record and Replay Troubleshooting for Networks**. accepted to *USENIX ATC 2011*, Portland, Oregon (to appear).

Mehmood, A., Wundsam, A., Uhlig, S., Levin, D., Sarrar, N., and Feldmann, A. **QoE-Lab: Towards evaluating Quality of Experience for Future Internet Conditions.** In Proceedings of *7th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom '11)*, (Location: Shanghai, China), April 2011.

Wundsam, A., Mehmood, A., Feldmann, A., and Maennel, O. **Network Troubleshooting with Mirror VNets.** In Proceedings of *IEEE Globecom 2010 Workshop of Network of the Future (FutureNet-III)*, (Location: Miami, FL, USA), December 2010

Earlier, extended version: Wundsam, A., Mehmood, A., Feldmann, A., and Maennel, O. **Improving Network Troubleshooting using Virtualization.** *Research Report Technische Universität Berlin, Fakultät Elektrotechnik und Informatik*, No. 2009-12, June 2009

Levin, D., Wundsam, A., Mehmood, A., and Feldmann, A. **BERLIN: The Berlin Experimental Router Laboratory for Innovative Networking.** In Proceedings of the *6th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom '10, poster session)*, (Location: Berlin, Germany), May 2010

Schaffrath, G., Werle, C., Papadimitriou, P., Feldmann, A., Bless, R., Greenhalgh, A., Wundsam, A., Kind, M., Maennel, O. and Mathy, L. **Virtualization Architecture: Proposal and Initial Prototype.** In *VISA 2009 - The First ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures*, August 2009

Manilici, V., Wundsam, A., Feldmann, A. and Vidales, P. **Potential benefit of flow-based routing in multihomed environments.** *European Transactions on Telecommunications (ETT)*, 20(7):650-659, 2009. (Invited paper).

Contents

Bevor ich morgens schnell bei Facebook reinguck,
hab ich keine Ahnung wie's mir geht.
Bevor ich morgens schnell bei Facebook reinguck,
weiß ich nicht, ob sich die Welt noch dreht.

Daniel Dickopf, Wise Guys, "Facebook"

The Internet is the first thing that humanity has built that humanity doesn't understand, the largest experiment in anarchy that we have ever had.

Eric Schmidt

1 Introduction

The two quotes above illustrate the fundamental dilemma of the Internet today. On the one hand: its growing importance. The authors of the first quote state that, before checking with Facebook in the morning, they hardly know how they are, or whether the world is still turning. In other words, users are increasingly relying on Internet services like Facebook, Twitter, Google, and Skype for their private and professional lives. The Internet is also considered a growing political factor. Efficient use of social networks has been cited as one of the success factors in the 2008 U.S. presidential election [128]. Social networks also have been named driving factors of the recent revolutions in the Middle East by popular news media, though the point is still under discussion [66, 143].

On the other hand, as stated by second quote from Google's then-CEO Eric Schmidt, the Internet as a whole still operates very much in ad-hoc fashion, and is not safe from operational failures, even by the people arguably at the center of its innovation. This was exemplified when a misconfiguration in Google's routing caused their services to be slow or unavailable on May 14, 2009 [149]. Other high profile network outages in recent years have included Pakistan Telecom trying to block Youtube for their customers, and inadvertently disrupting the service globally by announcing the prefix 208.65.153.0/24 on February 24, 2008 [126]. More recently, the overlay based communication network Skype faced a near-total outage from December 22, 2010 to December 23, 2010 [14]. This outage was caused by a combination of a regression and a faulty safety limit in the most popular version of their client software, leading to a domino effect of failing supernodes.

In April 2011, the US East region of the world's largest cloud infrastructure provider Amazon AWS [23] suffered a significant outage for 3 days, taking down several of the 200 world's most used web sites, e.g., Foursquare, Quora and Reddit, for significant time [24]. The problem was initially caused by an operator error during a routine maintenance. Overly aggressive recovery strategies in their network block-level storage systems then caused a rebuild storm and continuous destabilization. This affected the overall control plane of the region and in consequence the entire region, across several data-centers and "availability zones" designed to fail independently. In general, cloud offerings are perceived as valuable for their cost-effectiveness and scalability, but extremely challenging to develop against, because the end-host server has little insight into what happens and must check performance and validity through indirect measurements and protect against network-failure on the application layer.

This dilemma is likely to gain importance in the future, as more and more traditionally local computing services move to distributed, cloud based infrastructures. The availability and performance of such cloud services highly depends on our ability to precisely control, instrument and troubleshoot the networks that are involved.

For instance, it is very convenient to move a spreadsheet from your local computer to a cloud based service like Google Docs [75]. However, a user's ability to access this documents then depends on the availability and performance of a large number of networks. Specifically, packets and flows may have to traverse:

- The user's *residential home network*. Often, such small networks contain nothing more than a notebook and a commodity gateway router, so the average utilization is very low, but utilization at peak times may be high. Costs are very important.

- Alternatively, an *enterprise network* exposes different trade-offs for availability, cost and security.

- The *access line* to and *access network* of the user's Internet provider, often not redundant and a single point of failure.

- The *inter-provider Internet* with provider peerings, complex routing and the *core networks* of all involved transfer providers.

- The *datacenter network* in the service provider's computing center.

Keeping control of all these different environments and networks is no small task, nor is troubleshooting them when things go wrong. The wide range of scales and scenarios makes one-size-fits-all solutions difficult.

1.1 Our Approach

Due to the wide range of environments and scenarios outlined above, we do not strive for a single solution to all the challenges, but address each scenario individually and discuss several designs, architectures, and systems for improving our ability to troubleshoot and enhance our ability to control operational networks.

In this journey, we follow the rough chain of networks a flow may encounter on its way from the user to the cloud service. We start out in the residential context: With Flow Routing, we propose an approach that improves the reliability and performance of residential community networks (Chapter 3). We then turn to the inter-provider core network scenario and propose an Architecture for Network Virtualization (Chapter 4). In the same context, Mirror VNets (Chapter 5) improve troubleshooting and enable safe upgrades of globally networked services. Finally, turning to a customized datacenter or enterprise network scenario, we propose OFRewind (Chapter 6), the first system to enable practical record and replay debugging in Operational Networks.

The term operational networks also implies a degree of practicability; we want our approaches to work for real networks, not just on paper. This also means that we build prototype systems, and evaluate them in practical environments. It is our firm belief that, in the sense of Dave Clark [131], *running code* is an existential foundation for successful systems research. In our work, we strive for practical approaches that take into account the realities of networking and lend themselves to migration strategies.

We opt for a *network centric* approach. We believe that, while end-host centric troubleshooting and control techniques have been been intensively studied and improved substantially in recent years, the network as such has been under-investigated and underdeveloped so far.

More and more such services are moving to distributed, network based models, and subsequently have to deal with the failure modes and challenges inherent to networks. As such, concise instrumentation and troubleshooting abilities in the network will grow even more in importance.

1.2 Challenges for Network Troubleshooting

Why is network troubleshooting challenging and not a solved problem in practice? Some of the challenges are intrinsic to the characteristics of networks including their *distributed state*, their *speed*, *scale* and *geographical spread*. Other challenges are created by the characteristics of the prevalent *Internet architecture* and its base assumptions. Also, *organizational and economical* influences play an important role in steering (or, inhibiting) network innovation.

Challenges intrinsic to networks

A number of challenges associated with troubleshooting operational networks are intrinsic to the nature of networks themselves:

Distributed state: First, and foremost, a network is a *distributed system* with *distributed state* and *distributed configuration*. This means, that typically, no coherent view of the system/network state is available. Vantage points only provide a local view of the state. This can cause races and timer effects, and hampers observation of problems. In this context, the term *Heisenbug* has been coined in ironic reference to the *Heisenberg Uncertainty Principle* for problems that vanish or change when one tries to observe them.

Speed, scale, geographical spread: Due to the growing bandwidth demand, networks are often operated at the boundary of what is technologically feasible. Current enterprise links are moving to 10GB/s and providers are deploying 30 GB/s links. This means that there is little headroom for adding additional functionality, e.g., for instrumentation or troubleshooting. As previously discussed, networks operate at vastly different scales, from small residential networks to giant datacenter and operator networks. This makes it difficult to find abstractions that fit all of these environments. Also, the *geographical spread* of networks challenges our troubleshooting abilities — physical principles already limit the coherence achievable in a global provider network.

Internet architecture challenges

Other challenges for troubleshooting networks are created by the ubiquitous Internet architecture, and its basic assumptions, such as the *end-to-end principle* and the *KISS principle* of keeping the network "simple and stupid".

End-to-end principle challenges: The Internet architecture is built around an end-to-end principle, and aims to concentrate intelligence and state management at the end-hosts. This increases clarity and simplicity, facilitates end-host innovation and has allowed the Internet to grow from its feeble ARPANet beginnings to its current state. However, it also limits our troubleshooting abilities, as typically, the end hosts have very little insight into the conditions of the network. The consequent layering enables clear and simple interfaces and independent evolution of the lower layers, but it also hides valuable troubleshooting information from the components that depend on it.

KISS: In the spirit of the *KISS* principle [130], there is very little instrumentation support built into the architecture, e.g., for out-of-band notifications or performance isolation. Consequently, measurements and diagnostics often have to be attempted in-band. This increases the complexity of the higher layers and incurs the risk of *probe effects*, where the property under observation is inadvertently changed by the observation itself. There have been numerous proposals to improve the control and troubleshooting features of the Internet architecture and technology stack (e.g., DiffServ, Cross-Provider MPLS, etc.), but these have faced challenges in their adoption for organizational and economic reasons.

Overall, there exists a fundamental trade-off between the simplicity and clarity afforded by these principles and the limits in debugging and troubleshooting they impose. Approaches that aim at improving the instrumentation and control capabilities in the Internet architecture must consider the cost associated with the added complexity.

Organizational and economic challenges

Despite the intrinsic and architectural challenges outlined above, there have been several technologically promising proposals to improve the situation, like inter-provider MPLS [61], and DiffServ [108]. Most of these have gained only limited traction in practice though – a fate they share with other proposals for evolving the Internet architecture and core services like DNSSec, cBGP, and even IPv6[1]. This leads us to conclude that *organizational and economic* challenges play an important role for the practical adoption of innovations on the Internet.

Distributed administration: While some networks are controlled by a single operator, others are managed by more complex groups of operators, with relationships of differing trust levels. In virtualized, multi-tenant datacenters, a single owner controls the hardware, but the virtual hosts and networks are controlled by independent tenants with no trust relationship (or even knowledge) amongst themselves. The Internet in its entirety is controlled in a distributed fashion by roughly 37,000 Autonomous Systems[2]. As exemplified by the accidental Youtube hijacking by the Pakistan Internet Exchange, this system depends to a large degree on the correct behavior of its participants. Such complex administrative structures make ad-hoc debugging infeasible and require clear interfaces that provide controlled troubleshooting and instrumentation abilities, without disclosing unwanted information.

[1] Granted, it looks like 2011 may be the year that IPv6 will *finally* unfold.
[2] Announced AS numbers as of May 27, 2011 [118].

Economic hurdles to incremental adoption: Adoption of many useful proposals that enrich the classical best-effort Internet architecture with additional features useful for troubleshooting and improved control has been inhibited by economic hurdles. Often, a feature is not very useful until deployed by a large fraction of operators. Thus, nobody wants to move or invest first, keeping the Status-Quo at an "impasse" [28]. Other proposals violate the business interests of influential economic stakeholders (e.g., require operators to disclose details about their topologies) and consequently have small chances of adoption.

Summary

In summary, improving our troubleshooting and control abilities for Operational Networks faces a number of important challenges. Our approaches must cope with the fundamental properties of networks. Due to the wide distribution in scale and usage, a one-size-fits all solution may be hard to find, so we may have to investigate different solutions for different sizes and network characteristics. When challenging and adopting Internet architecture principles, it is important to consider the trade-offs this incurs for scalability and clarity. And it is important to consider the organizational and economic challenges faced by changes that require pervasive upgrades.

1.3 Guiding Principles

We now summarize the guiding principles of our work that follow from our basic approach and the challenges outlined in the past two sections.

Adapt to scenario: We investigate networks at different scales. A one-size-fits-all approach is difficult due to the vast range in scale and characteristics. Each approach should work well for a given scenario and network type.

Isolation: We want to limit the impact of our improved control and debugging abilities on the production traffic. Our observation should not introduce *probe effects*.

Deployability: Our approaches should consider the technological, organizational and economical conditions of the given scenario. We emphasize solutions that can be incrementally deployed.

Network perspective Our solutions must work inside the network. That means they must be transparent to the end-host and application level software.

1.4 Outline

We start out by discussing the background of our work. This includes a rundown of the status quo, the enabling technical innovations and descriptions of the basic testbeds and networks that have been utilized in our research.

We next turn to a local community network scenario. We propose *FlowRouting* as an approach available to residential communities of end users to self-improve their network connectivity and increase their available bandwidth at times of peak demand, and study its potential. Statistical multiplexing can be attractive as the average utilization of residential DSL lines has been shown to be low. We find that even simple load-balancing algorithms can significantly improve the user experience in times of peek demand, in particular for upstream traffic.

On a larger scale, we explore *Virtual Networks* (VNets) as a possible means to overcome the ossification of the Internet core, and propose an architecture for Network Virtualization in a multi-player, multi-role scenario. The proposed architecture differs from existing approaches in that it investigates the business roles, for example, leading to information hiding which is prevalent in a realistic multi-provider scenario.

With this architecture as a basis, we next turn to troubleshooting, and explore how the improved control afforded by VNets can benefit our troubleshooting abilities. We propose *Mirror VNets* as a primitive that enables safer evolution and improved debugging abilities for complex network services. Here, a *production VNet* is paired with a parallel *Mirror VNet* that is spawned in the same configuration and state as the original network. Then troubleshooting, or debugging can be safely attempted in the isolated Mirror VNet, without affecting the end users that interact with the production VNet. Only when a change has been verified to work, the VNets are *switched* semi-atomically. We present and investigate the approach, a prototype system built on OpenFlow and XEN, and discuss a case study in a multimedia/QoE scenario.

We then explore how flow-oriented architectures can be leveraged to enable replay debugging in Networks, and present and evaluate *OFRewind*, the first system to enable practical replay troubleshooting in Operational networks with black-box components, based on commodity hardware. OFRewind acts as a proxy in the OpenFlow controller chain, and enables consistent recording of traffic in a network domain. Control plane traffic is always recorded due to its low bitrates and high relevance for troubleshooting. Dataplane traffic is selectively recorded and can be load-balanced over multiple *DataStores* to keep hardware requirements within commodity bounds. The recordings can then be *replayed* back onto the network in coordinated fashion, with adjustable pace to investigate problem causes. We present several case studies where our tool has been instrumental in debugging practical network problems, and demonstrate it has adequate scalability and low enough overhead for practical use.

1.5 Our Contribution

Our contributions span a wide area in understanding and improving the control and troubleshooting of Operational Networks. In this thesis, we

- investigate the potential of *Flow-based-Routing* as a means to improve control over networks, and evaluate several flow-based routing algorithms, an idea that has recently gained main-stream momentum in the context of *Software Defined Networking* initiatives like OpenFlow [104].
- propose an *Architecture for Network Virtualization*, the first such architecture to incorporate aspects like information hiding that a prevalent in a realistic multi-provider scenario.
- propose *Mirror VNets*, which can enable safe upgrades and live troubleshooting for Internet services
- propose *OFRewind*, the first tool that enables coordinated record and replay for operational networks.

In lieu of a single comprehensive one-size-fits-all approach that necessarily would be complex and unspecific given the heterogeneity of of the environments, we propose network-centric approaches which are tailored to specific scenarios. As such, each solution is simpler, adapts well to the scenario, and consequently can be implemented and adopted more easily. In support of this claim, all approaches have been implemented on real systems and evaluated in operational networks. Consequently, we are confident that our approaches can serve as a data-point and guidance on how operational networks need to evolve in view of their increasing importance in our daily lives.

2
Background

In this section, we discuss the background of our work, in particular, recent innovations in Networking that act as technological enablers for our work: Virtual Networks and Software Defined Networks. We then present the testbeds that we use to evaluate our systems.

2.1 Virtual Networks

Virtual networks have recently been proposed as enablers for "overcoming the Internet impasse" [28]. Anderson et al. argue that easy access to virtualized testbeds can foster a renaissance in applied architectural research that extends beyond the currently incrementally deployable designs. For instance, clean-slate approaches that focus on full accountability or anonymity can be deployed inside of a virtual network, while leaving the current Internet untouched. Arguably, such testbeds can also provide an interesting platform for improving our control and troubleshooting abilities in networks.

Note that virtualization in computer systems in general is a rather old technique. It has been successfully applied to, e.g., CPU, memory, storage, and almost all other system resources. Recently, it has become possible to virtualize entire systems (hosts and routers) even on commodity platforms and with limited overhead. This enables running several isolated sets of programs or operating systems on a shared hardware. Also, *network links* have been virtualized using techniques such as VLANs and MPLS. These two techniques can now be combined to form entire *Virtual Networks*, where

entire virtual networks run isolation on shared hardware, in isolation from each other. As such, they may also span multiple physical or administrative domains.

Over the last years, virtual network architectures have been an area of active research. Some groups have focused on using network virtualization to enable larger-scale and more flexible testbeds. Other groups aim at virtualizing production networks or even the Internet.

Virtual Networks vs. VPN and overlays: Virtual networks differ from *overlay* networks in that they provide controlled visibility and control of the underlying network (the "underlay"). In contrast, overlays have no visibility into the network they are running on. For instance, they cannot modify the routing protocol used by the underlay, and can infer network properties only by in-band measurements. Classical *VPNs (Virtual Private Networks)*, especially MPLS-based Layer-2 VPNs, are similar to Virtual Networks, but typically do not include computing resources. Often, they are only available within the confines of a single operator network, and cross-provider connections require manual intervention.

We now introduce virtualization as a concept, then discuss system and link virtualization as the building blocks for virtual networks. We then discuss proposals for Virtual networks, and their applications and benefits, and the challenges that are under ongoing investigation.

2.1.1 Virtualization as a Concept: Properties and Benefits

Virtualization is an abstraction concept. A *resource* being virtualized (the *substrate*) is being made available by a *virtualization layer* to *virtualization guests*. In a general sense, this concept encompasses the aspects of *information hiding*, *multiplexing*, *isolation* and optionally *transparency*. Its benefits include *on-demand provisioning* and *migration*.

Information hiding: Virtualization abstracts and hides details of the underlaying layers from the *virtualized guests*. This can include, e.g., the exact address in physical memory where information is stored.

Multiplexing: Virtualization also typically encompasses an aspect of multiplexing: A single underlying resource is made available and multiplexed between several clients. For example, this is true for preemptive CPU scheduling in operating systems.

Isolation: Isolation is another important property that distinguishes virtualization from cooperative multiplexing. A virtualized guest should not be able to interfere with the operation of other guests running on the same substrate, nor must it interfere with the operation of the virtualization layer itself. Examples for this property include preemptive process scheduling in Operating Systems, where a process cannot avoid being unscheduled by the Operation System Scheduler. There are different levels of isolation that can be provided: *Functional Isolation* guarantees that there is no interference with the outcome of the computation. *Performance Isolation* guarantees that there is no impact on the performance of other guests.

Transparency: In some cases, virtualization also entails *transparency*. This means that the virtualization guest does not have to "know" it is being virtualized, but can instead continue to use the resource as if it fully owned it. Examples include virtual memory, where an application can typically rely on being able to transparently access a memory location, even when the accessed memory location has been paged out to disk. The relevant virtualization services (e.g., on-demand allocation, mapping to physical hardware, paging from/to background storage) are being transparently performed by the hardware or the operating system.

On-demand provisioning/overbooking: In many virtualization scenarios, it is possible to lazily provision resources only when they are actually needed, and thus "overbook" resources to optimize statical resource consumption. For instance, many system virtualization systems provide the option to provision storage only as it being allocated.

Migration: Another typical benefit of virtualization is that virtual resources can be quickly moved and/or transparently migrated between physical host resources. For example, virtual memory allows to change mapping between virtual and physical memory without affecting the (virtual) memory layout of the process. Some system virtualization products support live migration of virtual machines between physical hosts.

2.1.2 System and Link Virtualization

Individual resources have been virtualized in computer systems for a long time. More recently it has even become possible to virtualize full host systems on commodity hardware—a feature only offered by mainframe computers until then. Also, router vendors have started to offer virtualization support in their products. We now discuss *host sytem virtualization*, *router virtualization* and *network link virtualization* which form the fundamental building blocks for *Virtual Networks*.

Host virtualization: A relatively recent innovation, system virtualization has quickly become ubiquitous. It allows for better exploitation of the computing capabilities offered by contemporary commodity hardware, increases availability and eases management. Options range from *container based virtualization* products that run a single OS kernel and offer lightweight, but limited virtualization capabilities (VServer [153], OpenVZ [9], BSD jails) to *full virtualization* solutions that present a complete virtual PC hardware to the guest and allow for unmodified operation of guest systems (VMware [13], VirtualBox [152], KVM [90]).

Para-virtualization solutions, e.g., XEN [35], take a middle ground and allow several OS kernels to be executed in parallel under the supervision of a single hypervisor. These solutions do not fully emulate the PC hardware, but present a custom interface to the guests for communicating with the hypervisor. Consequently, they require changes to the guest operating systems, but potentially offer increased performance.

Note, however, that the distinction is not clear-cut: Even full virtualization solutions offer para-virtualized device drivers for improved I/O performance, and XEN offers a fully virtualized operation mode today. Also, the overhead of full virtualization has been reduced with the help of hardware virtualization interfaces, such as the VT-x technology from Intel [84].

Many full or para-virtualization solutions use a *hypervisor* or *virtual machine monitor*, a thin kernel running directly on the hardware that arbitrates access to the operating systems running inside of the virtual machines. Often, one such virtual machine is designated as *privileged* machine (also called *privileged domain* or *driver domain*). This machine is granted access to the system hardware to handle, i.e., network interfaces and storage devices. It also communicates with a protected hypervisor interface to control and monitor operations of the other virtual machines.

XEN, and many others, feature *live migration* support, in which a virtual machine is transferred to another physical host in powered-on state. They use incremental-delta transfers to minimize the downtime induced by the migration.

Some environments pose additional challenges for virtualization. In particular, this is true for real-time environments, where computation and/or exchanges of messages have to complete within bounded time. Specific virtualization architectures like Janus [127] take soft-real time constraints into account during scheduling, e.g., for Multimedia applications.

For most of our work, we choose XEN [35] as the host virtualization solution, because of its flexibility and performance, and for being a mature option in widespread use, while also freely available as open source.

Router/switch virtualization: Recently, vendors of network equipment have started to offer routers and switches that can be virtualized [86, 46]. Both software- and hardware isolated options are offered. These virtualization solutions aim at configuration and state isolation between multiple tenants sharing a physical router. Originally, they did not typically provide the ability custom *program* the devices to run custom routing protocols or network stacks. This has changed recently with the release of the JunOS SDK [81]. Virtualization solutions in this space have to consider the particular timing and scheduling constraints for network protocols, and provide appropriate precision.

Network link virtualization: Virtual links form the complementary building block for virtual networks. In local area networks, Ethernet VLANs are used to build virtual links. In port based form, they provide resource sharing and partitioning—a single switch can serve multiple isolated broadcast domains. In trunked mode, as defined by IEEE 802.1q, they enable multiplexing of several Layer-2 links onto a single link. Limited isolation can be achieved by using multiple queues and queue priorities. On the WAN scale, MPLS tunnels can provide a similar benefits. Until now, link virtualization solutions have lacked custom programmability. This is changed by the emerging concept of Software Defined Networks and OpenFlow, to be introduced in detail in the next section.

2.1.3 VNet Proposals for Experimental Networks

Virtualization plays a key role in creating flexible testbeds for Future Internet research.

PlanetLab family: PlanetLab [36] is a highly successful example of a distributed, large scale testbed. PlanetLab has a hierarchical model of trust which is rooted in *Planet Lab Central* (PLC). PLC is operated by the PlanetLab organization and is the ultimately trusted entity that authorizes access to the resources. Other actors are the *infrastructure owners* and the *users* that run their research experiments on PlanetLab. For each experiment virtual machines on various nodes are grouped to *slices* that can be managed and bootstrapped together. As the deployed virtualization mechanism offers only container based virtualization capabilities at the system level and do not virtualize the network stack, PlanetLab offers no network virtualization as such.

VINI, as proposed by Bavier et al. [150], is a testbed platform that extends the concept of virtualization to the network infrastructure. In VINI, routers are virtualized and interconnected by virtual links. As such, VINI allows researchers to deploy and evaluate new network architectures with real routing software, traffic loads, and network events. VINI supports simultaneous experiments with arbitrary network

topologies on a shared physical infrastructure. It builds on the architecture and management framework introduced by PlanetLab and extends the framework with interfaces to configure virtual links. The first implementation based on *User Mode Linux* [156] offers only limited performance.

An updated VINI platform, Trellis [38], allows for higher forwarding performance. It introduces a lower level system virtualization architecture that uses container based virtualization techniques for both system and network stack virtualization. Therefore virtualization flexibility is limited to the user space. VINI provides rudimentary concepts for end-user attachments [12] using OpenVPN tunnels and a single central gateway.

Downloadable distributions of the PlanetLab control framework and VINI are available as MyPLC and MyVINI, respectively.

Emulab: Emulab [53] also is a very popular testbed platform. Its offers a sophisticated management and life-cycle process but does not focus on the network architecture. Emulab provides virtual topology configuration based on ns2 configuration files and automatic bootstrapping of experiment nodes. Initially, Emulab focused on dedicated servers. Virtualization capabilities based on improved FreeBSD jails were added later.

GENI: GENI [70] is a large-scale U.S. initiative for building a federated virtualized testbed aiming at providing a powerful virtualized testbed for experimental purposes. Here, all operations are signed off and managed by a central *Geni Clearing House*. As a possible growth path, GENI plans on supporting federated clearing houses. During the first phases of the development both—VINI/PlanetLab and Emulab—have been used as GENI prototypes.

All testbed oriented architectures mentioned above do not consider several key factors relevant for virtualizing the (commercial) Internet: They assume a hierarchical trust model that centers on a universally trusted entity, e.g., the PLC/GENI clearinghouses.

2.1.4 VNet Proposals for Production Networks

We now discuss existing proposals for Production Networks and the commercial Internet as a whole.

CABO [63] proposes to speed up deployment of new protocols by allowing multiple concurrent virtualized networks in parallel. To this end, infrastructure providers manage the substrate resources while service providers operate their own customized network inside of allocated slices of these networks.

This idea in refined by Cabernet [161] which introduces a "Connectivity Layer" between the roles mentioned above. This layer is responsible for splicing the partial networks provided by the Infrastructure Layer and the presenting them as as a single network to the Service Layer. It facilitates the entry of new service providers by abstracting the negotiations with different infrastructure providers and allows for aggregation of several VNets into one set of infrastructure level resource reservations.

2.1.5 Challenges and Ongoing Work

Several challenges associated with Virtual Networks are still under active investigation:

Performance of virtualized software routers: Several groups have shown that it is possible but challenging to build high performance software routers of commodity hardware. For example, Egi et al. [59] show that in XEN, the privileged domain is able to forward packets at near native speed whereas XEN unprivileged domains offer very poor performance. As such, they conclude that for virtual router platforms all data forwarding should be done in the privileged domains while the unprivileged domains should be restricted to the control plane. Bhatia et al. [38] investigate several mechanisms on how to improve VINI's forwarding performance on commodity hardware, and identify the Linux bridge as a performance bottleneck. Several authors investigate parallelism strategies for optimizing performance on multicore software routers [54, 58] and identify cache hierarchy coherence as crucial. Anwer [30] propose an NetFGPA-based virtualized NIC card for optimized performance.

Management of virtualized networks: Automated management interfaces are an important aspect of virtual networks. This includes the questions of resource allocation for virtual networks and the question of when to move which virtual router and which virtual link to which physical resource. Good algorithms can reduce downtime and therefore increase availability and stability. A nice side effect can be improved energy efficiency. The challenges are

1. to find good ways of specifying virtual networks such that the specification entails sufficient degrees of freedom. Example approaches include the GENI *rspec* language [71].
2. to find good algorithms for embedding the specified virtual networks onto the physical substrate given the fluctuations in the demands for virtual networks and outages in the physical substrate. Yu et. al. [160] suggest to improve virtual network embedding by adding substrate support for path splitting and migration. Chowdhury et. al. [45] propose an algorithm with coordinated Node and Link Embedding. Yeow et. al. [159] focus on reliability aspects by providing diversification and pooling of backup resources.

3. to find efficient methods for migrating virtual links and virtual routers without downtimes. The VROOM (Virtual ROuters On the Move) [154] architecture proposes such live virtual router migration and remapping of virtual links. Bienkowski et al. [39] provide an optimal offline algorithm for service migration and competitive migration strategy.

4. to be able to manage the above process efficiently and debug it. This is one aspect addressed by this thesis.

In summary, Virtual Networks combine system and link virtualization techniques. They are considered enablers for scalable experimentation with and incremental deployment of network innovations. A number of challenges, including performance, management and debugging are still under active investigation.

2.2 Software Defined Networks / OpenFlow

As discussed in the last section, virtualization in network components has so far often lacked the programmability aspect: Virtualized routers typically only provide configuration isolation between multiple guests. Ethernet VLANs and MPLS tunnels are available, but also provide limited customization. It is possible to build custom network components from commodity PCs in software, e.g., [59], or from flexible FPGA based experiment hardware like NetFPGA [95], but these solutions are constrained in performance, scale and cost and power efficiency. For instance, PC servers require high performance CPUs for multi-Gigabit dataplane forwarding that are expensive and consume significant power. Typically, a network optimized server can provide about 6-8 Gigabit network ports per rack height unit. In contrast, a commodity switch provides full cross-section bandwidth between its ports, and typically provides 48 ports per height unit.

Recent years have seen increased interest in custom programmable, scalable network hardware, driven by the requirements of large scale data-centers, researcher seeking scalable experimental evaluation, and Internet providers.

Scalable experimental evaluation: Researchers have put forward many new proposals for improving the Internet, e.g., in Future Internet initiatives [15]. It has been recognized that these proposals are not widely adopted unless they can be practically evaluated at scale. This has been a problem in network research so far. Custom hardware is typically prohibitively expensive to build for a research project, while software solutions often do not scale to large enough systems. This drives the demand for testbed components that share some of the properties of commercial commodity hardware (high line-rates, high fan-out, affordable pricing) with some of the flexibility afforded by software based testbeds (ability to try custom protocols, quickly innovate).

Data centers: Another push for increased flexibility in Operational Networks comes from the operators of big data centers. These data-centers often contain several 10,000 servers and experience highly differentiated, specific network workload patterns (e.g., induced by the Map-Reduce pattern). As such, their operators are highly interested in flexibly controlling their networks enable optimal resource utilization, and effectively reducing their operational costs and gain a competitive advantage in the market. It has been shown that standard networking protocols and mechanisms (e.g., flood-based ARP, Spanning Tree) are no longer sufficient [88].

Network operators: Network operators have also voiced interest in standardized programmability of network devices for reducing their dependency on specific hardware vendors, reduce operation costs, and improve their changes of diversification [146].

An emerging class of network architectures addresses these issues. They are called *split forwarding architectures* that enable *Software Defined Networks*. Examples include OpenFlow [104], Tesseract [158], and Forces [5]. These architectures *split* the forwarding of traffic on the dataplane off from the decision making about the forwarding, which happens on the control plane. In doing so, they enable custom programmability and centralization of the control plane, while allowing for commodity high-throughput, high-fanout data plane forwarding elements. OpenFlow is by far most widely used of these architectures, and is discussed in detail here.

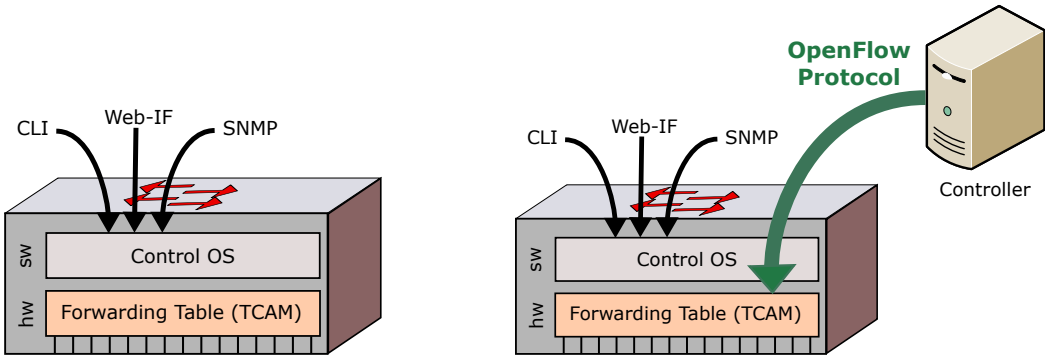
2.2.1 Overview of OpenFlow

OpenFlow is an open protocol that enables custom programmability of the control plane of an Ethernet switch. Originally introduced by the Cleanslate Lab of Stanford University [140], and developed by a mixed academic-industrial consortium, it is now managed by a recently formed non-profit organization, the *Open Networking Foundation* [112].

The main idea of OpenFlow is to add a standardized programmable API to a commodity Ethernet switch. This interface exposes the forwarding table of the switch to an external *controller*. OpenFlow *controller* applications can then assume control of the forwarding process by issuing flow-based commands. Thus, they can, e.g., optimize traffic flows for load-balancing or security policy. The actual forwarding, is still done in hardware on the switch.

Consider the architecture of a classical "smart" or enterprise switch, as depicted in Figure 2.1(a). It typically consists of:

1. At least one forwarding table, typically built out of TCAM (Ternary content-addressable memory). This flow table can map incoming packets to output



(a) Classical switch (b) OpenFlow-enabled switch

Figure 2.1: Comparison of schematic switch architectures with and without OpenFlow

actions in constant time. Historically, switches based their forwarding exclusively on the Destination MAC address, but recent models also support other attributes, like VLAN tags and even higher layer attributes like IP-addresses.

2. This flow-table is managed by an embedded system running on the switch, that installs and removes entries. This system is typically configured via CLI, SNMP or a web-based administration interface.

For an OpenFlow switch, as seen in Figure 2.1(b), another interface is added to the embedded system that allows direct, fine-grained control over the flow table via the OpenFlow protocol. To this end, the traffic is grouped into *flows*. Each flow can be associated with specific actions, that cause its packets, e.g., to be directed to a specific switch port, or on a specified VLAN. The flow definition can be tailored to the specific application case—OpenFlow supports a 11-tuple of packet header parts that can be matched on, ranging from Layer 1 (VLAN ports), via Layer 2 and 3 (MAC and IP addresses) to Layer 4 (TCP and UDP ports).

This separation facilitates implementing novel routing, switching or traffic management approaches. To this end, operators or experimenters can write a piece of user-space controller software, and run it on commodity PC acting as controller for OpenFlow switches. Note that the actual data-plane forwarding is still performed by the switch and at line-rate. Open Flow thus potentially enables a whole range of new research and engineering approaches, including cheap routers, flexible middle box platforms, monitoring and debugging and traffic engineering.

2.2.2 An Example of an OpenFlow Message Exchange

We now discuss an example of the messages exchanged for setting a up a flow in an OpenFlow-enabled network, as shown in Figure 2.2. At the center are the OpenFlow-

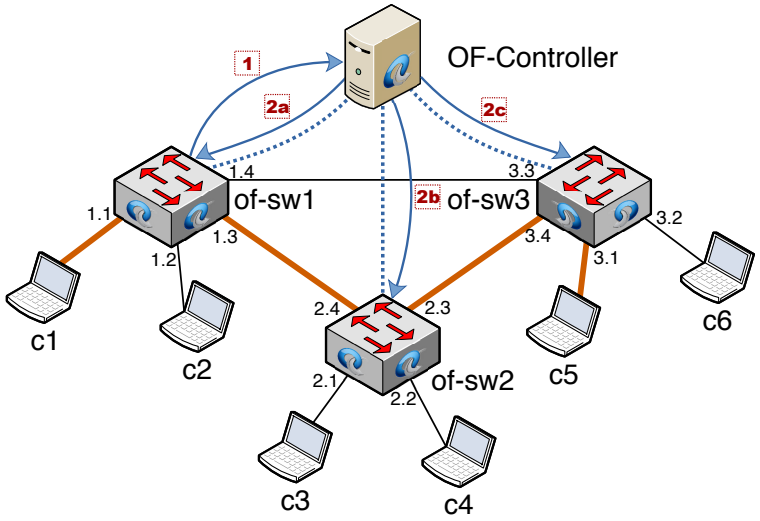


Figure 2.2: Example of a message exchange in an OpenFlow-enabled network

enabled switches `of-sw1`, `of-sw2`, and `of-sw3`. They are all managed by the controller `ofctrl` via the OpenFlow protocol (blush dotted lines)[1]. Each of the switches has two attached clients, called `c1` through `c6`.

Initially, the only preset rules are for ARP. Now `c1` wants to communicate with `c5`. `c1`'s first packet triggers, on arrival at `of-sw1`, an OpenFlow `PACKET-IN` message which is sent to the controller, depicted by the arrow marked (1). If the controller decides to instantiate a flow it sends a `FLOW-MOD` message to the switches `of-sw1` (2a), `of-sw2` (2b), and `of-sw3` (2c). The `FLOW-MOD` message consists of a *match* and an *action* part. The *match* part is responsible for selecting packets going from `c1` to `c5` by some means (e.g., source and destination IP addresses or MAC addresses). The *action* part directs the matched packets to port 1.3 (message 2a), 2.3 (2b), and 3.1 (2c), respectively. Thus, packets from this flow are sent from `c1` via `of-sw1`, `of-sw2`, and `of-sw3`. The reverse direction is either setup independently when packets arrive at *of-sw3* for `c1` from `c5`, or, alternatively, the controller can decide to setup this path proactively. When the flow becomes idle or times out, the switch removes the entry from the flow table and sends a `FLOW-EXPIRED` message to the controller. This message contains summary statistics about the completed flow.

2.2.3 Existing OpenFlow Controllers

We now give an overview of popular existing OpenFlow controllers, as summarized by Table 2.1:

[1]There is no fundamental restriction that limits the deployment to a single controller.

Table 2.1: Overview of popular OpenFlow controllers (as of 05/2011)

Controller	Language	Category	Plugin Model	Protocol
SimpleController	ANSI-C	Technology Demo	-	1.1
NOX	C++ / Python	Controller framework	C++/Python modules	1.0
Beacon	Java	Controller framework	OSGI components	1.0
FlowVisor	ANSI-C*	Virtualization Controller	ANSI C modules	1.0

* A newer version is based on Java.

OpenFlow SimpleController: There is a simple controller implementation available within the OpenFlow reference implementation. It mostly serves as a technology demo and baseline reference. By default, it behaves similar to a learning switch: When an unmatched packet hits the controller, a hash table of MAC-addresses to port mappings is checked. If the destination MAC address is known to be attached to certain port, a flow is installed that outputs the packet, and all subsequent ones belonging to this flow to the correct port. Otherwise, the packet is flooded to all attached ports. The hash-table is updated on every `PACKET-IN`.

NOX: NOX [110, 76] is the most widely used controller implementation. It is free software published under the *GNU Public License (GPL)* and has been developed by Nicira [7].

NOX has been described as an "Operating System for networks". It is designed as an extensible, event-based controller framework. The actual decision-making is carried out by custom modules ("applications") which can be implemented in Python or C++. Based on a publisher/subscriber model, NOX monitors the network and informs the interested applications about relevant events, e.g., appearing and disappearing switches or received packets (new flows).

Beacon: Beacon [37] is a Java-based OpenFlow controller framework, licensed under the Apache 2.0 license. It is built as a set of OSGI components. This allows the OpenFlow modules on the platform to be installed, started, or stopped at run-time, without disconnecting switches.

FlowVisor: FlowVisor [135, 136] is special purpose controller aimed at Network Virtualization. It acts as a proxy between several OpenFlow switches and controllers. Logically, the switch is partitioned into separate *slices*, with each of the client controllers being able to see and influence a part of the possible set of flow entries (the so-called *flow space*). It aims at providing transparency and isolation between the slices. In order to provide good isolation properties, FlowVisor needs precise information about the performance impact of individual flow commands (e.g., whether a flow table entry will be handled by software or hardware). At present, such information is not available, so it has to resort to heuristics.

Table 2.2: Overview of popular OpenFlow switch implementations (as of 05/2011)

Implementation	Type	Target	Protocol	# flows (exakt/wildcard)
Reference	Software	Reference Platform	1.0	65536[§] / 100[§]
OpenVSwitch	Software	Product	1.0	$*$ / 0[†]
NetFPGA	Hardware	Experimental Platform	1.0	32,000 / 100
Broadcom	Hardware	Reference Platform	1.0	2,048[⋆]
HP Procurve 5400 Series	Hardware	prototype for product	1.0	1500[⋆]
NEC IP 8800	Hardware	supported as product	1.0	3072[⋆]

[⋆] Mixed flow table – can contain both wildcarded and non-wildcarded flows
[§] Software limit – can be changed through recompilation
[‡] OpenVSwitch handles all wildcard flow entries in user-space and adds caching exact flow entries into kernel space

2.2.4 Existing Switch Implementations

In this section, we discuss present several current OpenFlow switch implementations, including two software based implementations (*Reference*, *OpenVSwitch*) and four hardware based implementations (*NetFPGA*, *BroadComRef*, *HP*, *NEC*). Table 2.2 gives an overview. Other companies have announced plans to support OpenFlow.

OpenFlow reference implementation: A reference implementation of OpenFlow is maintained by Stanford university. It contains a software switch implementation built in user-space[2]. Running on various Linux-based devices in the network, this currently is the most widely used switch implementation of OpenFlow. It has also been ported to run on OpenWRT based wireless access points.

OpenVSwitch: OpenVSwitch [113, 115] is another software based switch. It is an advanced reimplementation of the standard Linux software bridge to bring it en par with, e.g., Sun's *ProjectCrossbow* for Solaris [144]. Features include support for OpenFlow, VLANs, SFlow/NetFlow and channel bonding. It is developed by Nicira, and released under GPL. It aims to be production quality software. It supports OpenFlow version 1.0.

Broadcom reference implementation: Based on the Broadcom Firebolt chipset, a reference hardware switch implementation is built by the consortium. According to the consortium, it is "targeted for research use and as a baseline for vendor implementations, but not for direct deployment" [134]. It supports version 1.0.

HP Procurve: HP has a prototype implementation of OpenFlow for their ProCurve series of switches. It currently supports version 1.0.

[2] A kernel based implementation has been dropped due to resource constraints.

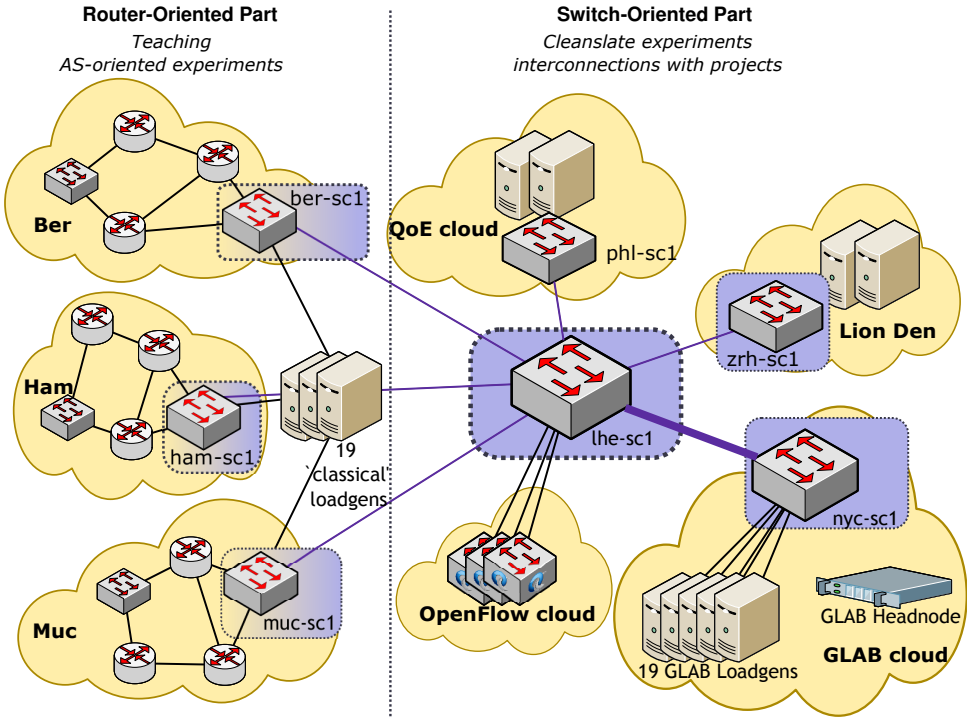


Figure 2.3: Schematic Network Layout of the FG INET Routerlab

NEC IP 8800: NEC offers the model IP 8800. It provides line-rate support for OpenFlow, and is supported as a product. It, too, supports OpenFlow version 1.0. Other production switches have been announced.

2.3 Testbeds

We now present the experimental networks and testbed platforms we use for parts of early practical evaluation.

2.3.1 FG INET Routerlab / BERLIN

The Routerlab is the experimental network platform of the research group INET in Berlin. It contains commodity switches, routers, load-generating servers, and network path emulators (e.g., for emulating DSL access line characteristics) from multiple vendors. To reflect the Internet's structure on a miniature scale, its devices are organized into several, hierarchical and semi-isomorphic clouds that are connected using a variety of different electrical and optical connections, depicted in Figure 2.3. This allows an experiment to comprise emulated end-user access lines, edge level routers, AS'es, and a high speed forwarding core, using real hardware.

The testbed includes 43 commodity rack servers with 2-8 cores, routers from Cisco and Juniper, and switches from Cisco, HP, NEC, Quantas. Special purpose NetFPGA cards are available at a subset of servers. The Routerlab features a *hybrid*, *customizable* physical topology. One part of the testbed is organized in a *router-centric* fashion for teaching, experimentation with commercial routers and current routing protocols (depicted in the left third of Figure 2.3). The other part is *switch-centric*, with devices fully meshed onto a manageable switch fabric with 200+ ports. This part is mainly used for research experiments with novel network architectures and technologies, but also switched interconnections to other testbeds and projects.

Labtool—the Routerlab management platform: This varied landscape of devices is managed by our custom software management system, called *Labtool*. *Labtool* presents a unified, vendor-agnostic interface to the experimenter for device reservation, configuration, interaction (e.g., console access, power management), and topology management. It also maintains a complete and historically versioned picture of the physical and logical network testbed topology. Labtool is integrated with an automated system configuration and disk imaging tool which allows disk images and router and switch configurations to be deployed quickly onto arbitrary experimentation devices. Labtool is experiment-centric, in that it organizes all of its functionality around the management, configuration, and repeatable deployment of experimental topologies and device configurations. The software architecture of the Labtool utilizes a three-layer client, server, and database structure, and is built to be extendable and scriptable with a client API in Ruby. Labtool provides the following functionalities:

Experiment life cycle management: The Labtool maintains an experiment-level view of all the actions it performs. This means that devices, the physical and virtual links connecting them, and their individual configurations are kept in the underlying database schema. This allows for easier hibernation, migration, and restoration of any particular experimental setup.

Physical topology versioning: The Labtool keeps track of all custom cabling changes over time and across experiments. Versioned cabling enables QoE-Lab administrators to alter and reliably restore topology changes.

Boot-mode configuration with imager system: An experiment performed on one set of devices should be repeatable on another set of suitably similar devices. To this end, the Labtool allows experimental device configurations for a given device to be redeployed onto any sufficiently similar device. The Labtool provides a collection of hardware-agnostic base operating system images which facilitate quick deployment of experimental environments.

BERLIN—pluggable services for the Routerlab: The unified management interface provided by the Labtool allows the Routerlab to offer pre-configured combinations of

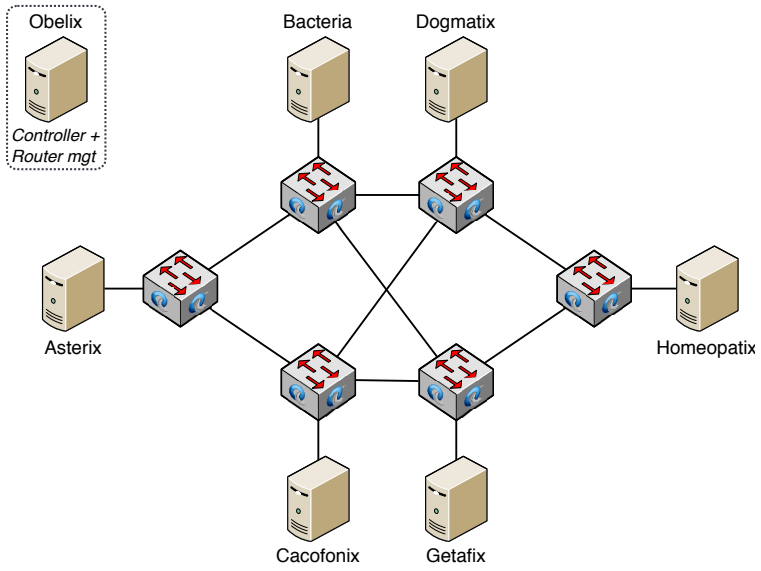


Figure 2.4: Schematic network layout of the Los Altos Testbed

hardware and software as higher-level *pluggable services* within the BERLIN framework. These fulfill many common requirements, e.g., traffic generation, monitoring and capturing, network emulation, NetFPGA packet processing, and virtualization services. These pluggable services allow researchers to quickly establish an experimental setup with most of the required services from pre-built components. For instance, an experimenter may want to evaluate a new router primitive implemented as a *NetFPGA* program, then require *self similar background traffic* to be generated and routed through the *NetFPGA*, apply emulated WAN line delay characteristics, and finally capture packet level traces at several points in the experiment.

2.3.2 Los Altos Testbed

We also take advantage of the OpenFlow testbed of T-Labs Los Altos[3], depicted in Figure 2.4. The Los altos testbed consists of 6 OpenFlow-enabled switches with 48-96 ports from 3 different vendors and 7 commodity 8-core servers with 2-16GB RAM and 500GB to 2 terrabytes of harddrives. As the testbed is not as heavily co-shared as the Routerlab, the devices are managed in an adhoc fashion, and the topology is adapted as required.

[3]Officially Deutsche Telekom Inc., R&D Lab USA

2.4 Summary

In this chapter, we introduce two important enabling innovations that form the basis of our work: Virtual Networks and Software Defined Networks, as exemplified by OpenFlow, both provide us with powerful tools to control and troubleshoot networks. We use the Virtual Networks concept when proposing our *Architecture for Virtual Networks* in Chapter 4, as well as in the our proposal of *Mirror VNets* (Chapter 5). We use the notion of Software Defined Networks and OpenFlow for *Flow Routing* (Chapter 3), for implementing *Mirror VNets*, and for *OFRewind*, presented in Chapter 6.

Also, we present the testbeds used for early practical evaluation of our work. In particular, the flexible *Routerlab* platform is used throughout our work, except the specific case studies in OFRewind.

3

Augmenting Commodity Internet Access with Flow-Routing

We start our journey following the flow on its way from the end-user's laptop to a cloud datacenter server in the residential context, and target commodity Internet access lines. Due to the broad adoption of broadband DSL and cable lines and NAT routers—often offered as contractual gifts by Internet Providers [98], we can assume that the first network a flow encounters is likely to be a small wireless network, followed by the user's commodity WLAN router and her DSL or cable based broadband line. In a similar fashion, commodity Internet access lines have also become popular for small and medium businesses due to their low costs and relatively high bandwidths [137].

Why do these two network hops pose challenges relevant to our mission? Well, typically, they are not redundant. Hence, they represent a single point of failure for the user's ability to access cloud services. Also, they can present a performance bottleneck for the user's experience. This is especially true for upstream traffic, as the upstream bandwidth is typically an order of magnitude smaller than the advertised downstream bandwidth in in many consumer-targeted Internet offerings. This can cause problems, e.g., for HD video conferences, cloud based backup services like Dropbox [55] or uploads to social media sites. In effect, the data rates provisioned by broadband Internet access connections can continue to fall short of the requirements posed by emerging applications. Thus, it is attractive, e.g., for a small business, to *combine* several commodity access lines to optimize performance and availability. Moreover, the average utilization of a residential Internet connection has been shown to be low [98, 111]. This indicates that there is potential to statistically optimize

the performance and availability of residential Internet access by *sharing* Internet access lines between users in a residential community. Due to prevalence of flat-rate tariffs for end-users, no financial compensation may be required for such a scheme. In summary, statistical multiplexing of user traffic across multiple broadband access connections over a shared wireless medium can be used to improve availablility, increase the peak data rates, and satisfy high instantaneous traffic demand of bandwidth-intensive applications.

However, combining or sharing commodity Internet lines is not trivially achievable, as realities of current network infrastructure design and operation pose challenges. For instance, commodity Internet customers do not have the option to act as a multihomed autonomous system and participate in inter-AS routing to leverage Internet Routing techniques for availability and load-balancing. Also, ISPs do not support connection bundling in their commodity offerings. This limits the extent to which statistical multiplexing can be leveraged for bandwidth aggregation. To overcome these limitations, we propose *flow-routing* to route one user's flows via a wireless shared medium through another user's broadband connection. We have presented a first prototypical flow-routing system in an earlier work [157], which we refine in this chapter with the help of Software Defined Networks and OpenFlow. Also, several other multihoming mechanisms have been proposed by other authors, e.g., [129, 147]. Complementary to these efforts, our present work investigates the potential for flow-routing and attempts to answer a more fundamental question: what is the *attainable* benefit of flow-based routing in such multi-homed environments?

To answer this question, we employ hybrid simulations and real testbed experiments in various settings, using both synthetic and measurement-based traces for representing user traffic. We propose a flow-level simulator, *FlowSim*, which simulates various flow routing algorithms based on actual flow traces from real networks as input, and use a testbed using commodity wireless home gateways and DSL links. Synthetic Web traces generated using Surge [34] and residential access traces from the Munich Scientific Network (MWN) serve as traffic input. Our results show that for bandwidth intensive flows the average download time can be reduced by up to a factor of 3. This emphasizes the substantial potential benefit of flow routing among commodity broadband connections. To summarize, this work makes the following contributions:

- A flow-routing scheme that enables the combination of several commodity Internet access lines for increased reliability and performance.
- A flow-based simulator and an experimental setup for evaluating flow-routing: reproducible results are obtained and the difference between ideal and progressively realistic experiments is quantified.
- Several flow-routing strategies, ranging from idealized ones, in which full knowledge of links and flows is assumed, to a practical and realizable one that is suitable for implementation on a router.

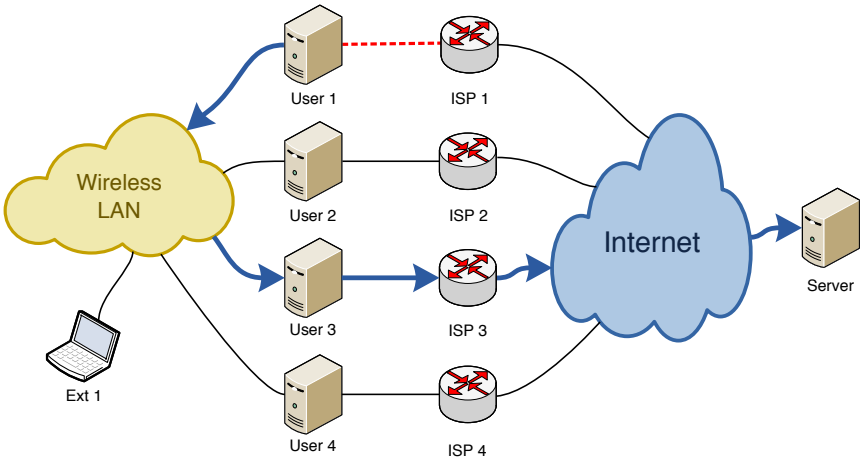


Figure 3.1: Example of a community network with Flow-Routing

- An evaluation of the benefits and potential of the various flow routing strategies in multi-homed environments using synthetic and real traffic loads.

3.1 Flow-Routing Approach

Consider the scenario shown in Figure 3.1. It depicts a community of users accessing the Internet via DSL links provisioned by different ISPs that share an ad-hoc wireless LAN. User 1 may experience congestion on her connection due to a large persistent upload. However, if User 3's Internet connection has spare bandwidth, User 1 may be able to redirect her traffic to User 3's connection via multihoming.

We now describe the earlier prototype flow-routing system, then propose a refined implementation based on OpenFlow [104].

3.1.1 Earlier Prototype: FlowRoute

In earlier work we developed a prototype, named `FlowRoute` [157], that enables flow routing on the client machines. It consists of three main components: a preloaded shared library, called `libConnect`, which intercepts all client access to the Berkeley socket layer and delegates routing decisions. `Routed`, the unique central decision making module, has a global view of the link loads and makes the actual routing decisions based on the flow routing strategies previously described. The third component on the client machines, called `Proxy`, is responsible for re-directing the flows between a client and its destination and for informing `Routed` about the load of all shared DSL connections.

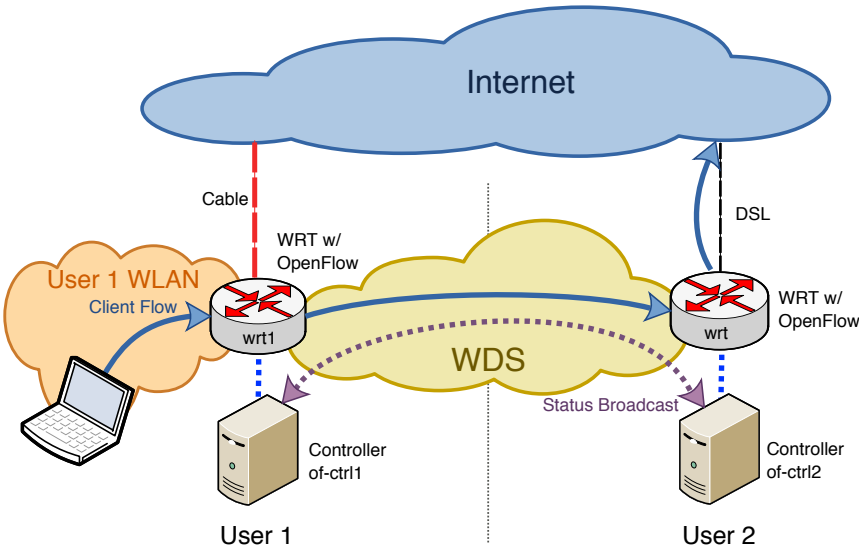


Figure 3.2: Components of a Flow-Routing system with OpenFlow

3.1.2 OpenFlow-Based Flow-Routing

Refining our approach proposed in prior work [157], we use OpenFlow for implementing flow-routing[1]. Figure 3.2 shows how Flow-Routing is performed between two users, User 1 (shown in the left half on the Figure) and User 2 (right half). Both users have commodity broadband Internet access lines, User 1 has a cable Internet connection, User 2 a DSL connection. Both users use a commodity WLAN router with OpenFlow-enabled firmware installed, e.g., a Linksys WRT [132]. An OpenFlow controller local to each network serves as the flow-routing manager. In our research context, we use a controller running on a commodity PC connected to the router via wired Ethernet. For a production environment, the controller process may alternatively be deployed directly onto the router. Consider a situation where a new client flow is started at User 1's notebook, but her local cable line is congested due to other traffic. When arriving at User 1's router (*wrt1*), the new flow triggers a `PACKET-IN` message to her controller (*of-ctrl1*). *Of-ctrl1* determines that the local line is congested, while the line attached to *wrt2* is empty. Accordingly, it decides to reroute the flow via User 2, and sends a corresponding `FLOW-MOD` message to *wrt1*. The flow is thus redirected via a wireless network that spans the residential community, e.g., using WDS [155]. When the flow arrives at *wrt2* and causes a `PACKET-IN` at User 2's controller *of-ctrl2*, it is admitted, and a `FLOW-MOD` is sent that routes the flow to User 2's DSL line. Every controller monitors the utilization of its local line by `FLOW-STATS` requests in regular intervals, say, every 5 seconds. They also keep track of the number of flows admitted to the local line at any given moment. This status

[1] See Section 2.2 for an introduction of the OpenFlow protocol and messages.

information is regularly broadcast to all other controllers via the WDS network.

We now discus, in Section 3.5 related work. In Section 3.2, we introduce our experimental approach and the simulator, and describe our testbed. Simulator and testbed results using measurement-based and synthetic Web traces as input are discussed in Section 3.3. We then summarize our findings in Section 3.6.

3.2 Methodology

To evaluate the attainable benefit of flow-based routing in multihomed environments, we combine simulations with real testbed experiments. For our evaluation we derive the workload from user traces from the border router of the Munich Scientific Network to capture real user behavior. However, since it is difficult to capture the reaction of the applications by only replaying traces we also rely on synthetic workloads generated by the Web traffic generator Surge [34]. This allows us to quantify potential performance improvements as experienced by the users.

3.2.1 Flow Routing Strategies

To obtain a baseline regarding the potential benefits of flow routing we investigate the following set of basic flow routing strategies.

Direct: the current practice for residential broadband connections: every flow is routed on the direct link, i.e., the DSL connection of the user originating the flow.

FatPipe: the "ideal case"; it bundles all DSL links into one fat pipe with bandwidth equal to the sum of the capacities of the individual connections. The only restriction is that no flow may exceed the bandwidth of the originating DSL connection. This strategy cannot be implemented in a real network and is only supported by the simulator.

MinLargeFlows: a "first algorithm"; it tries to minimize the number of bulky flows that share any of the DSL links. Therefore, it assigns each new flow to the link which, at the time the flow arrives, carries the lowest number of flows that have already transmitted some number of bytes, say 8 KB.

Moving flows from one link to another is difficult as it requires either support by all end-systems or transparent movement of the TCP state. Therefore, we do not allow in any algorithm re-routing a flow once it has been assigned to a certain link.

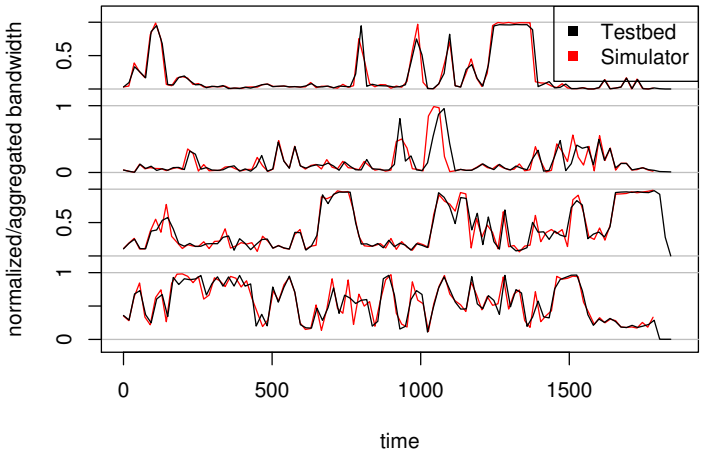


Figure 3.3: Comparison simulator vs. testbed: Normalized link utilization (direct routing)

3.2.2 Flow Routing Testbed: FlowRoute

A testbed for conducting realistic experiments has been deployed at our site, emulating the setup in Figure 3.1. Four nodes have been deployed on one floor in our building, coexisting with other wireless networks, as in real residential scenarios. The distribution of the testbed nodes ensures wireless connectivity among them. Each node consists of a wireless router with IEEE 802.11a/b/g interfaces and a client machine which is also equipped with a wireless interface. The routers are directly connected to the Internet via 2 Mbps DSL lines and to the corresponding client machines using an Ethernet interface. The client based flow routing system `FlowRoute` allows us to evaluate the benefits of flow routing in real-world testbeds. Hence, effects not easily captured in a simulator, such as the impact of different shared interconnection mediums, wired vs. wireless Ethernet, etc., can be studied.

For some of the experiments we need to flexibly tune the properties of the access lines, e.g., the delay, drop parameters and bandwidth. For these experiments, we replace the DSL lines with the NistNet network emulator [109]. The NistNet emulator is installed on a separate machine that is accessible by all testbed nodes via the management network.

3.2.3 Simulator: FlowSim

We have developed a simulator, called `FlowSim` [101], to evaluate in a scalable manner the potential of flow routing strategies using flow-level traces as input, captured from real networks or testbed experiments. Popular packet level simulators such as ns-2 or SSFNet are not suitable for our purpose, given the large number of flows considered here. To validate the simulator we reproduce the testbed experiments within `FlowSim` and compare the results.

We can use `FlowSim` to simulate network setups such as those shown in Figure 3.1, with various number of users, different capacities and delays for uplink and downlink directions and flow routing algorithm parameters. The interconnection capacity between DSL subscribers is assumed not to be a bottleneck and thus modeled as infinite. As such, the simulator does not distinguish if the community is interconnected via a wireless or wired network.

`FlowSim` consists of two main components: a *router* and a *scheduler*. The *router* identifies interactive and bandwidth-limited flows. Bandwidth-limited flows are further classified by the direction (referred to as dominant direction) in which they exceed a threshold into downlink-limited, uplink-limited, and both-limited flows. Once a flow has been classified, it is routed using one of the flow routing strategies.

The role of the *scheduler* is to distribute the available bandwidth among competing flows, while approximating the dynamics of the TCP protocol. In order to achieve reasonable performance, it uses a fluid TCP model operating on discrete time slots, rather than on a per event basis.

The simulation results do not show a significant difference when using smaller time slots than 1/10 second, the current setting. The distribution of the available up and downstream bandwidth is done in a fair manner between all flows that use one link. Each bandwidth-limited flow exercises a TCP-style slow start before it is able to attain its fair share of the bandwidth. While connect and close delays are modeled, packet losses are ignored. Comparisons with results obtained from the testbed for the same set of traces used as input show that the approximations are reasonable. For each bandwidth-limited flow its fair share of the bandwidth is computed based on its dominant direction(s). The bandwidth share for the non-dominant direction is then set in proportion to the transmitted bytes. The results are then scaled to the available bandwidth.

As an initial validation of the simulator we, in Figure 3.3, compare the link utilization's from a simulation run with that of an experiment run in the testbed. Each subplot shows the normalized bandwidth usage across time for both the simulator (in red color) as well as the testbed run (in black color). With the exception of a few outliers and some lags, the curves match closely. This indicates that the simplifications within `FlowSim` are reasonable.

3.3 Results

To evaluate the potential benefit of flow routing we explore its behavior both in the simulator as well as in the testbed under different workloads. We start with a Web workload generated by Surge. Then we switch to a trace-based evaluation relying on traces from the Munich Scientific Network.

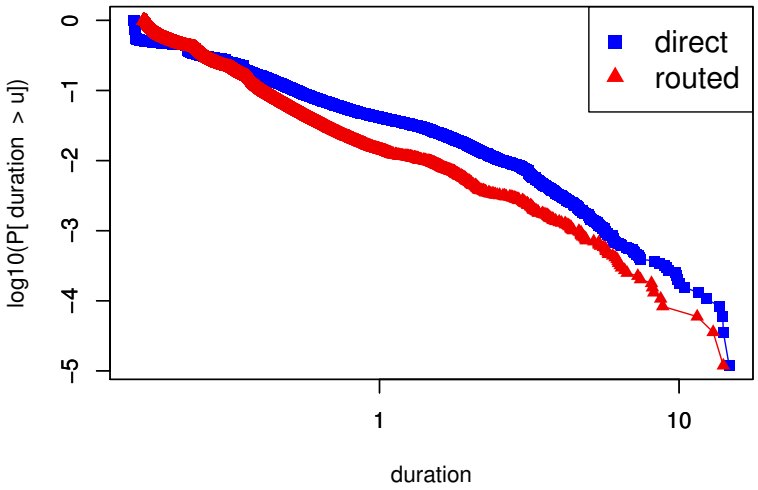


Figure 3.4: Surge Experiment: CCDF of flow durations without vs. with Flow-Routing (MinLargeFlows strategy).

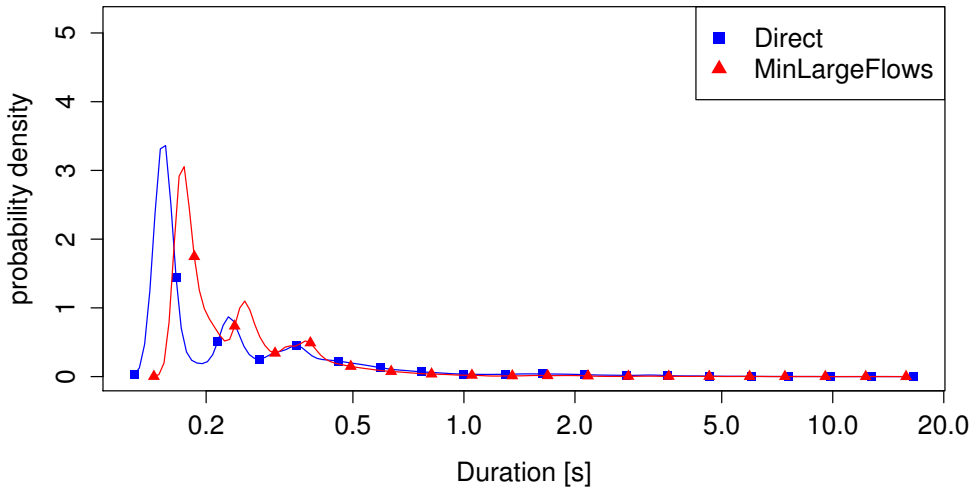


Figure 3.5: Surge Experiment: PDF of flow durations without vs. with Flow-Routing (MinLargeFlows strategy).

Experiment	*Policy*	*Mean*	*Median*
Surge (Ethernet, NistNet)	MinLargeFlows	2.47	1.83
Surge (Ethernet, DSL)	MinLargeFlows	2.49	2.03
Surge (Wlan, NistNet)	MinLargeFlows	2.61	2.11
Surge (Simulator)	MinLargeFlows	2.59	2.09
Surge (Simulator)	FatPipe	2.79	2.24
1/3 MWN (Ethernet, NistNet.)	MinLargeFlows	2.04	1.02
MWN (Ethernet, NistNet)	MinLargeFlows	3.02	1.98
MWN (Simulator)	MinLargeFlows	2.47	1.99
MWN (Simulator)	FatPipe	3.73	2.40

Table 3.1: Flow-Routing experiments overview: benefit for bulky flows (>0.5 sec.) under different workloads and experimental settings for both simulator and testbed.

3.3.1 Synthetic Web Workload

We use Surge [34] to generate a synthetic workload that resembles Web traffic. We update its configuration parameters to reflect characteristics of today's Web (e.g., the median and mean HTTP object size of the current Web workload mix, including Web 2.0 and P2P, as observed in the MWN traces). We use the popular Apache2 software as our Web server. To impose a reasonable load we use four Surge instances per household (host). This Web workload results in an average utilization of 0.39 Mbps per DSL link.

Testbed Results

Figure 3.4 displays the *complementary cumulative distribution function* (CCDF) of the flow durations on a log-log scale for an experiment with NistNet and a wired interconnection network for the flow routing strategy `MinLargeFlows`. One can clearly see the benefit of flow routing for longer flows, as durations with flow routing are shorter than without. The corresponding logarithmic *probability density function* (PDF) is shown in Figure 3.5, highlighting that the flowrouting system imposes some overhead for short flows.

To quantify the achievable benefit, we compute the ratio of the durations, more precisely, we compute *duration(direct) / duration(routed)*. Larger values denote better performance under flow routing. Considering all flows, the mean benefit is 1.11 with a median of 0.90. The benefit is small due to the prevalence of short flows in this scenario, which all suffer from the overhead of our prototype flow routing implementation. When we only consider flows with a duration larger than 0.5 seconds the

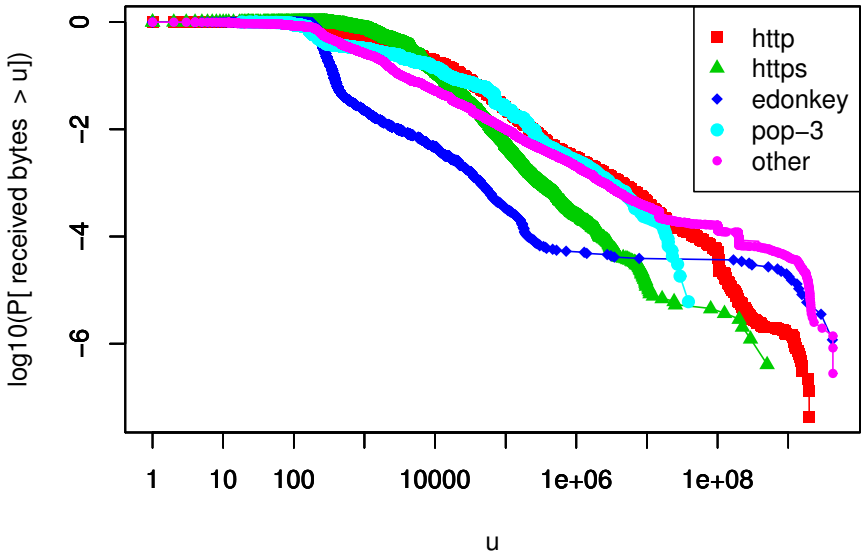


Figure 3.6: Analysis of the MWN trace: CCDF of received bytes per application.

mean improvement increased to 2.47 (median 1.83). This implies that there is a significant benefit for bulky flows.

We next switch from NistNet to using the actual 2 Mbps DSL lines, with the wired interconnection network. We observe that the results improve slightly in comparison to the emulated network. For all flows, the mean benefit is 1.20 (median 0.95). Flows that last longer than 0.5 seconds experience an improvement of 2.49 (median 2.03). The differences are explainable by small inaccuracies in NistNet.

Finally, we use the well connected wireless network in 802.11a mode as our interconnection network within the community. Somewhat surprisingly, the overall results improve slightly when compared to using a wired network. The mean (median) improvements for longer flows are 2.61 (2.11) and for all flows 1.27 (1.0). This shows that the wireless network is not the bottleneck. We reroute only 75% of the flows which imposes a load of less then 5 Mbps on the wireless network. Overall, we see that significant improvements are possible by taking advantage of multihoming.

Simulator Results

We investigate the upper bound of the routing performance by using the simulator on the traces gathered from the Web workload experiments. This allows us to estimate the potential benefits and compare the performance prediction of the simulator against the performance improvements observed in the testbed. The simulator displays some deviations in flow duration, which are in part due to the simulator's assumption that bandwidth is shared fairly between flows. This assumption is known

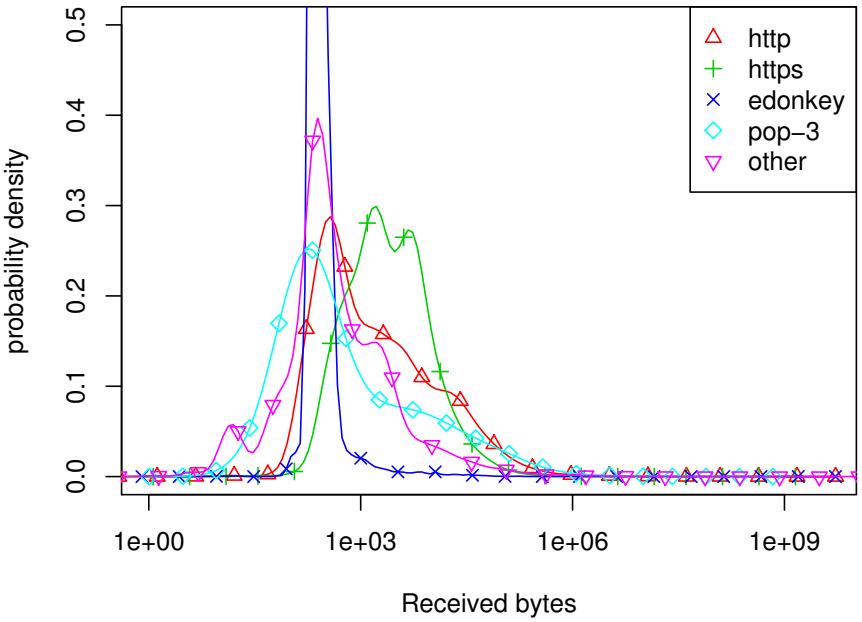


Figure 3.7: Analysis of the MWN trace: PDF of received bytes per application.

to work well on average but not on smaller time scales. As such, it is not surprising that the size of the deviations decrease as flow durations increase. In total, the ratio of the durations for bandwidth-limited flows in the simulation vs. the testbed experiment has a mean of 1.07 and a median of 0.97. This indicates that, while for individual flows the predicted performance does not agree with the performance seen in the experiment, the overall results of the simulation match the experiment quite nicely.

We then compare the performance as predicted by the simulator with the performance of the actual experiment using the `MinLargeFlows` routing policy. The simulator reports a mean (median) improvement for flows longer than 0.5 seconds of 2.59 (2.09). In the synthetic Web workload experiment with the same strategy, the inter-arrival times between Web connections are shorter than with the `Direct` policy, and the load is also higher. These predictions match the actually observed benefits of 2.47 very well. Interestingly, the predicted mean improvement by the simulator for `FatPipe` is only slightly better than for `MinLargeFlows`: with a mean improvement of 2.79 vs. 2.24. This confirms that the achievable benefit for this scenario is indeed limited by the traffic properties of this specific Web workload.

3.3.2 Trace-Based Experiments

Using traces of real traffic to drive simulations as well as testbed experiments allows repeatability of results under realistic loads.

Workload

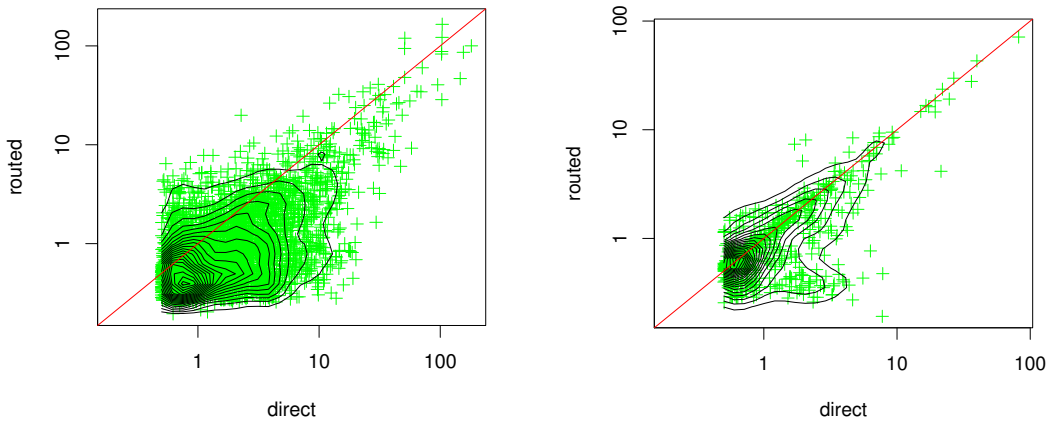
We use connection-level summaries of traffic traces captured at the border router of the Munich Scientific Network, MWN. The MWN provides 10 Gbps Internet upstream capacity to roughly 50,000 hosts at two major universities including student dormitories, with the daily traffic amounting to 3-6 TB. Ample bandwidth is available to all users of this network via high-speed local area networks. Thus, Internet usage is not impaired by any bandwidth limitations of the access links.

In a typical 24 hour workday trace from April 24th 2007 we identify approximately 37 million flows, out of which 21.1 million have incomplete connection information records, a typical component of todays traffic mix, such as SYN attempts (56%), rejected connections (27%), in progress connections (8%), and missing information (9%). For our experiments, we consider the remaining 15.9 million flows, through which 641 GB (182 GB) were transferred from (to) the Internet. These volumes correspond to an average bandwidth utilization of 60 Mbps (17 Mbps) downstream (upstream).

To better understand the characteristics of the traffic, we classify the flows according to the application that most likely generated them, as identified by the port number. About 73.50% of the flows are HTTP, 7.83% HTTPS, 2.74% eDonkey, 0.52% POP-3, and 15.41% are other traffic. Figure 3.6 shows the CCDF (complimentary cumulative distribution function) of the received bytes for different applications on a log-log scale. We observe that the flow sizes of the applications are consistent with the heavy-tailed distributions that have previously been reported, e.g., [50]. The behavior of eDonkey may seem strange at first, but this is due to its two traffic components—the control channel and data transfer connection. Part of the byte contributions of other P2P file sharing protocols are captured by the "Other" class; hence, it is not surprising that the tails of the two curves coincide. The other P2P traffic is contained in the "HTTP" class. The mean number of bytes over all connections is 43,409 and the median 795. Similar observations hold for the bytes sent per flow.

To investigate the actual flow size distributions, we plot the PDF (probability density function) of the logarithm of the transfer sizes on a logarithmic X-axis, in Fig. 3.7. HTTP/ HTTPS exhibits the expected distribution with a small spike that corresponds to the typical header size. HTTPS has a larger mean but a smaller median. POP3 transfers are smaller, while eDonkey is on the extreme end with many short connections due to control traffic and several larger ones due to data transfers. The "Other" category is a mix of unidentified flow types, which also seems to contain a significant amount of P2P traffic.

From this large collection of flow data we selected a subset of flows, referred to as *MWN* flow trace, that originate from IP addresses that are assigned to student residences, to ensure that we consider only genuine residential traffic. Students are provided Internet access only via 28 NAT gateways. The traffic via those NAT



(a) Full MWN trace (b) 1/3MWN trace

Figure 3.8: MWN trace experiment: Scatter plots of flow durations without vs. with Flow-Routing (MinLargeFlows strategy). A superimposed contour plot shows the density of occurrences.

gateways imposes a load of about 0.34 Mbps on the upstream and 5.74 Mbps on the downstream. We assigned these 28 NAT gateways to our DSL links randomly. Using such a dense trace allows us to investigate times with peak traffic while keeping the durations of our experiments relatively short (30 minutes).

To quantify the performance of our flow re-routing against different traffic loads, we run experiments with the `MinLargeFlows` strategy on the testbed using the full MWN trace in addition to a random subselection of 1/3 of the total number of flows (reduced set).

Performance Analysis

Given that the short flows suffer from the overhead imposed by the specific implementation chosen for our prototype we again concentrate on bulky flows – those that last longer than 0.5 seconds. Figures 3.8(a) and 3.8(b) show scatter plots of flow durations. Each point represents a flow, with the duration of the direct routing policy on the X axis and the duration of the re-routed policy on the Y axis. The overlaid contour lines plot the density of measurement points in one region. More points below the $x = y$ diagonal imply more flows that benefit from routing. The density peak for the experiment with the full MWN trace is significantly below the diagonal, whereas the peak for the reduced MWN trace is closer to the diagonal.

The experiment shows improvements with respect to the flow durations: for the former case a mean factor of 3.02 (median 1.98) and in the latter case by a factor of 2.04 (median 1.02). When considering all flows including the short ones, the

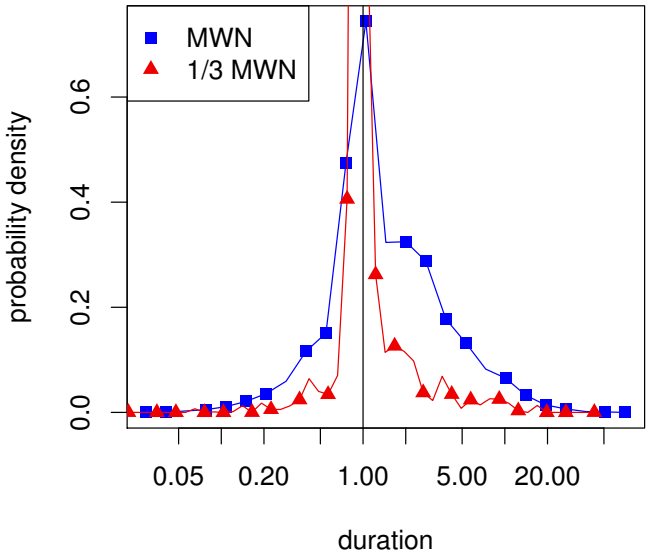


Figure 3.9: MWN trace experiment: PDF of flow duration ratios with different workloads

mean/median values are for the full MWN trace 2.21/1.12 and 1.17/0.94 for the reduced set. See Table 3.1 for a summary.

We plot the logarithmic density (PDF) of the flow durations ratio, calculated as $duration(direct)/duration(routed)$ in Figure 3.9. We note that the shift towards right (values larger than 1) is much more pronounced for the full trace experiment. This means that flow routing performs better in regions close to the link congestion. Figure 3.10 shows the CCDF of flow durations with and without flow routing using the `MinLargeFlows` strategy. We observe that the flows exhibit significantly shorter durations when flow routing is used.

The experimental results in the testbed for the standard `MinLargeFlows` routing algorithm compare well with the simulation. The median ratio of flow durations is 1.98 in the testbed and 1.99 in the simulator. Mean ratio of 3.02 is better in the testbed than 2.47 achieved in the simulator. This is partly due to the larger jitters that we observe in the testbed. The `FatPipe` policy achieves flow duration improvements by a mean factor of 3.73 (median 2.40) in the simulator.

Next, we explore the effect of the shared interconnection medium, wireless or wired. For this purpose we rely on the testbed. Average and median flow durations vary by less than 15% between a wireless shared medium with good connectivity and a wired one. The DSL emulation through NistNet achieves less than 3% average and median variation in flow lengths. Figure 3.11 compares the density distribution of the flow durations when using DSL lines or NistNet for the Internet connection and Ethernet or Wireless for the testbed interconnections.

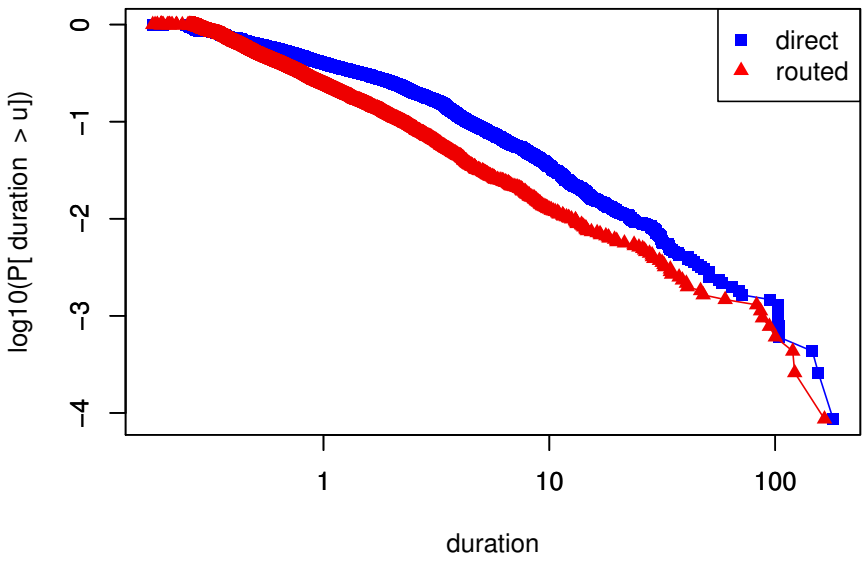


Figure 3.10: MWN trace experiment: CCDF of flow durations with vs. without routing

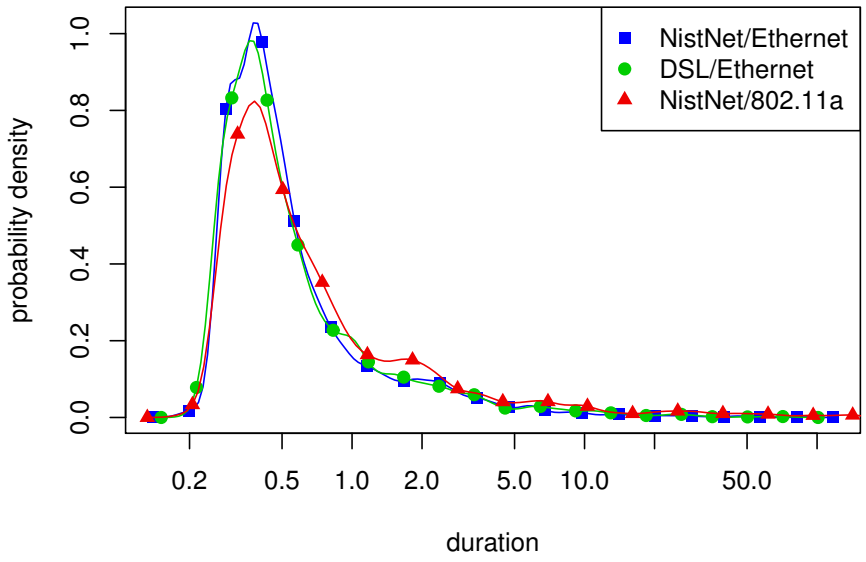


Figure 3.11: MWN trace experiment: PDF of flow durations on different network media.

3.4 Discussion

From our results, we observe that Flow-Routing can enable communities to self-improve the reliability and performance of their Internet access lines. Especially during periods of congestion and for bulky flows, performance gains can be significant. We now discuss some limitations of the usability of this scheme.

3.4.1 Legal Issues

When Internet access lines are shared, legal responsibility for the use of the Internet access line can be problematic, as the contractual customer may be held responsible for violations, e.g., of copyright laws, committed by users that share his access. Approaches to this problem include *limitation of forwarded traffic*, *non-repudiable cryptographic logs*, and *IP separation by the provider*.

Limiting forwarded traffic: Participants may choose to only forward certain types of traffic on behalf of other users, e.g., excluding Bit-Torrent and NNTP traffic. However, the usefulness of this approach deteriorates, as HTTP hosters like RapidShare [98] are increasingly used for distribution of media content of questionable origin.

Cryptographic logs: The Flow-Routing system can be augmented by cryptographic means to provide non-repudiable signatures on forwarding requests. This can help a user to prove who was the original requestor of certain traffic.

Provider integration If the Internet provider partakes in the distribution scheme, traffic from different users can be assigned different IP addresses and thus legally attributed to different users. This is a clean solution and liberates the sharing user from responsibility for the forwarded traffic, but requires collaboration from the provider.

3.4.2 Fairness

Fairness can become an issue in a Flow-Routing community, when users utilize their access lines to a different degree. This can be addressed by augmenting the Flow-Routing system with a compensation scheme,e.g., a maximum cap on the inbalance of traffic forwarded between two participants. Local traffic can be preferred over traffic on behalf of other users. Forwarded traffic can be prioritized based on the account balance. We leave further investigation of such schemes for future work.

3.4.3 Interconnection speed

As the speed of commodity Internet access lines continues to improve, the wireless interconnection between the participants may become the bottleneck that limits the performance of forwarded traffic. This may be avoided by wired interconnections, e.g., Gigabit Ethernet, between the participating routers – still an order of magnitude faster than the highest-bandwidth available commodity Internet offerings. Alternatively, one can limit the amount of traffic that is forwarded. This can be achieved by restricting forwarding to situations of significant congestion or failure of the local access line.

3.4.4 Unaffected scenario

Issues with legal aspects or fairness and the capacity of the interconnection medium can reduce the viability and performance of Flow-Routing when access lines are shared over a wireless link between multiple entities. We have discussed approaches to limit the impact of these issues. Note, however, that there is a useful scenario for Flow-Routing in which they do not apply at all: When a single entity, e.g., a small business, uses Flow-Routing to take advance of a diverse set of access lines for improved reliability and performance. This may indeed be the most viable and useful use case for Flow-Routing, and increasingly important, as the diversity of Internet Access solutions available to the end customer continues to increase.

3.5 Related Work

The areas of local community networking and connectivity sharing have been well explored and some enterprise solutions [18, 72, 77] have even been deployed in the real world. Bundling connectivity by utilizing multiple links in parallel is often used in access networks as well as backbones, where it is commonly known by the term "channel bonding". The mechanism assumes that the links terminate at the same devices.

Commercial efforts similar to flow routing and targeted to centrally-administered enterprises use the name *smart routing*. Smart routing is marketed by a number of companies, including Internap [85], Radware [120], Angran [25], Mushroom Networks [107] and Viprinet [151]. The proposed algorithms, independent of the general Internet routing, focus on cost in addition to network performance. Goldberg et al. [73] analyze the performance of smart routing and its potential both for interfering with BGP and for self-interference. The authors of [51] show that smart routing can bring economical benefits even to the ISPs.

Akella et al. [18] find that multihoming has the potential of improving throughput performance by up to 25% compared to using only the best ISP connection. In addition, Akella et al. [17] report on an upper bound for the possible performance improvements with multihoming. It is roughly 40%. They also show that a careful choice of providers is necessary. We show that even greater improvements are possible during congested periods, and for bulky flows.

Closely related to flow management, MAR [129] provides a framework for leveraging the diversity of wireless access providers and channels to provide improved wireless data performance for the commuters via a wireless multihomed device that can be deployed in moving vehicles. Their packet scheduling architecture requires the presence of a proxy server as a concentrator and support from the ISP side.

The MultiNet project [43] proposes a multihomed single-card device by means of virtualization. In MultiNet the wireless card is continuously switched across multiple networks. In contrast to physically accessing multiple networks, our work brings forward the idea of routing flows across the network in order to achieve better performance.

The *Stream Control Transmission Protocol* (SCTP) is a layer-4 protocol that supports multihoming intrinsically. It has been proposed by Stewart [142] and documented by RFC 4960 [141]. SCTP was primarily designed for Public Switched Telephone Network (PSTN) signaling messages over IP and still has to see wide deployment into popular TCP/IP protocol stack implementations. The address management takes place during the setup of the association and cannot be changed later.

pTCP [83] is an extension of TCP that enables bandwidth aggregation for multihomed devices. The protocol has been evaluated through simulations. Its mechanism is based on packet buffering and appears to us to require a large management overhead.

Habib et al. [80] propose the resurrection of the session layer for striping data from a single connection over several links. Implementing the proposal requires extensive changes at the OS or the application level, which is also a requirement for SCTP or the IPv6 shim6 layer [31].

Recent work by Thompson et al. [147] evaluates a framework for end-host multihoming with flow routing based on RTT measurements and prediction of the flow sizes. The evaluation is limited to a proof of concept system consisting of two nodes and one specific flow routing algorithm. Another example of flow management is the work presented by Tao et al. [145]. They study the feasibility of improving performance by increasing path diversity through flow-based path switching. This work was evaluated using a wired experimental setup but no evaluation in wireless environments has been reported.

Papadopouli and Schulzrinne [114] propose the integration of channels with largely different QoS characteristics for serving streams with adaptable bandwidth. Their

prototype operates as a multicast application with variable bandwidth. Standard applications such as WWW and email are unable to take advantage of the proposed system. Lad et al. [91] propose a high-level architecture for sharing connectivity in a coalition peering without discussing its realization or presenting a performance evaluation.

Complementary to the above approaches, the work reported in this chapter offers an evaluation of multihoming in wireless environments. There are a plethora of papers that have reported on the difficulties posed by wireless environments and their impact on urban networks, e.g., or [41]. They indicate that careful planning of the wireless domains and avoidance of congested channels are necessary to achieve good wireless interconnection results.

3.6 Summary

In a residential or small business context, a commodity broadband Internet access line is the first "hurdle" encountered by the end user. This concerns reliability as well as performance. Reliability and customer services of such commodity offerings have been reported to be problematic [102]. The performance of such lines, though adequate for mean utilization, can negatively impact the user experience during peak demand: Especially social media interaction or cloud backup services increasingly put demands upstream capacity. Upstream bandwidth in such offerings is an order of magnitude lower than the downstream bandwidth.

Flow-based routing can enable the combination of multiple commodity lines for improved reliability and performance. It also enables residential end-users to share their Internet lines in a residential community and profit from statistical multiplexing and the low average utilization of residential Internet access lines.

In this work, we propose a refined flow-routing system based on OpenFlow. We note that the effort required to deploy such a system drops significantly due to the easily programmable dataplane offered by OpenFlow. Furthermore, we evaluate several flow re-routing strategies for their performance characteristics using synthetic workloads and captured user traces as traffic input. Our encouraging results show improvements by up to a factor of 3 in the download times of bandwidth-limited flows, with very small overhead. Bandwidth-demanding flows and an unbalanced distribution of load among the users of the sharing community particularly benefit from the flow-based routing approach. More importantly, full knowledge of flow characteristics is not needed, as indicated by the performance of `MinLargeFlows`.

4

A Control Architecture for Network Virtualization

Leaving the residential and access context, we now turn to the core of the Internet and inter-provider issues. Note that while the Internet still aptly fulfills its current mission as a packet network delivering connectivity service, it was also designed with assumptions that no longer describe all current and future requirements. Stronger security, better mobility support, more flexible routing, enhanced reliability and robust service guarantees are some examples of areas where innovation is needed [64]. More precisely, while the network itself has indeed evolved tremendously in terms of size, speed, new sub-IP link technologies, and new applications, it is the architecture of the public Internet that has mostly remained the same. In a sense, the Internet is a victim of its own success as its size and scope render the introduction and deployment of new network technologies and services very difficult. In fact, the Internet core can be considered to be suffering from "ossification" [48], a condition where technical and technological innovation meets natural resistance.

Apart from the technical challenges involved in evolving such a large and important system, the Core Internet and its inter-provider space are in the most challenging environment to for another reason as well: because it is a "no man's land" of responsibility. Here, a connection between a user and a service provider can very well by affected by problems in the network of any provider on the path, or in the interconnection points between any of the providers. Often neither the *end user* nor the *service provider* has a direct contractual relationship to the provider causing the problem, and thus no economic handle to force the trouble-causing party to solve the problem. For instance, in 2008, Level-1 provider Cogent unilaterally stopped peering

with their competitor TeliaSonera [100]. This caused reachability and performance problems for end-users routed via Cogent to services hosted by TeliaSonera, e.g., the European servers of the popular online game World Of Warcraft. Yet, neither the company offering the service (Blizzard Entertainment) nor most of the affected end customers were direct customers of Cogent.

This showcases that improving control and troubleshooting in this complex environment is as much an organizational, economical, and political challenge as a technical one. The challenge is most prominently exemplified by *Quality Of Service* (QoS). It is widely believed that end-to-end QoS is required to guarantee good user experience for demanding applications. Mechanisms have been researched, proposed and standardized for more than a decade (e.g., DiffServ, cross-provider MPLS). Still, these new enabling technologies fail to achieve traction across the majority of ISPs. Consequently, we believe that the greatest challenge is not in finding solutions and improvements to the Internet's many problems, but in how to actually deploy those solutions and re-balance the tussle between reliability and functionality.

Network virtualization provides a promising approach to enable the co-existence of innovation and reliability. The type of network virtualization needed is not to be confused with current technologies such as Virtual Private Networks (VPNs), which merely provide traffic isolation: full administrative control, as well as potentially full customization of the virtual networks (VNets) are also required. This also enables non-IP networks to be run alongside the current Internet realized as one future virtual network. Each of these virtual networks can be built according to different design criteria.

In this chapter, we present a network virtualization architecture for a Future Internet, which we motivate by analyzing business roles. In contrast to the GENI initiative [70] our goal is not to provide an experimental infrastructure but to identify the key roles and stakeholders that are necessary to offer virtual network based services across the Internet. We identify four main players/roles, namely the Physical Infrastructure Providers (PIPs), Virtual Network Providers (VNPs), Virtual Network Operators (VNOs) and Service Providers (SPs). This re-enforces and further develops the separation of infrastructure provider and Internet service provider advocated in [63, 161]. Indeed, we will show that the architecture encompasses other proposed network virtualization architectures.

This virtual network architecture is specifically designed to enable resource sharing among the various stakeholders, thereby increasing its adoptability. In today's Internet, ISPs as well as service providers (e.g., Google) are continuously searching for opportunities to either increase revenue or to reduce costs by launching new services, investing in new technology (CAPEX) or by decreasing operational costs (OPEX). To understand the order of magnitude of the investment cost consider that AT&T plans to invest 17–18 Bn $ in 2009 [2] compared to a revenue of 124 Bn $ in 2008 [3] and Deutsche Telekom invested 8.7 Bn € compared to revenues of 62 Bn € in 2008 [1].

Thanks to increased resource sharing, even a modest reduction in the investments of, say, 1% can result in several millions of savings per year.

The analysis of business roles is presented in Section 4.1. Section 4.2 provides details of the virtual network architecture. We discuss the benefits and challenges of Virtual Networks in Section 4.3. In Section 4.4 we give an overview of related work in comparison to our approach. We summarize our findings in Section 4.5.

4.1 Virtualization Business Roles

The major actors in the current Internet are service providers (e.g., Google) and Internet Service Providers (ISPs). Hereby, an ISP offers customers access to the Internet by relying on its own infrastructure, by renting infrastructure from someone, or by any combination of the two. Service providers offer services on the Internet. In essence, ISPs provide a connectivity service, very often on their own infrastructure, even if they also lease part of that infrastructure to other ISPs. For example, AT&T and Deutsche Telekom are mainly Internet Service Providers, while Google and Blizzard, and Skype are Service Providers.

Despite this "dual-actor landscape" [63, 161], there are already three main business roles at play in the current Internet: *infrastructure provider*, which owns and manages an underlaying physical infrastructure (called "substrate"); *connectivity provider*, which provides bit-pipes and end-to-end connectivity to end-users; and *service provider*, which offers application, data and content services to end-users.

However, the distinction between these roles has often been hidden inside a single company. For example, the division inside an ISP that is responsible for day-to-day operation of the network is rarely the one that is planing and specifying the evolution of the network.

By identifying these players we can on the one hand identify different business opportunities and on the other hand disentangle the technical issues from the business decisions.

When considering the kind of network virtualization that enables the concurrent existence of several, potentially service-tailored, networks, a new level of indirection and abstraction is introduced, which leads to the re-definition of existing, and addition of new, business roles:

Physical Infrastructure Provider (PIP), which owns and manages the physical infrastructure (the substrate), and provides wholesale of raw bit and processing services (i.e., slices) which support network virtualization.

Virtual Network Provider (VNP), which is responsible for assembling virtual resources from one or multiple PIPs into a virtual topology.

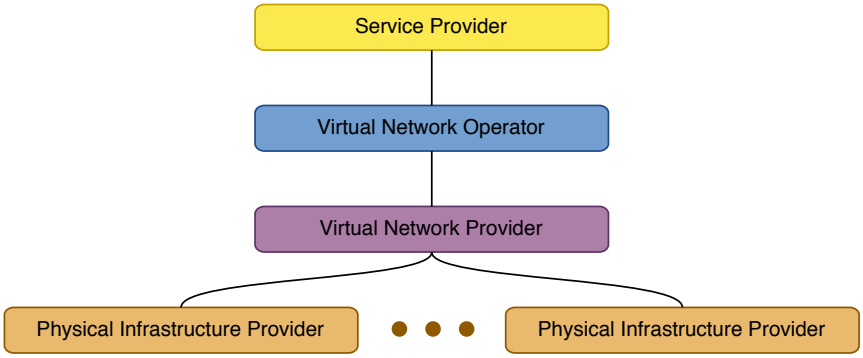


Figure 4.1: VNet management and business roles

Virtual Network Operator (VNO), which is responsible for the installation and operation of a VNet over the virtual topology provided by the VNP according to the needs of the SP, and thus realizes a tailored connectivity service.

Service Provider (SP), which uses the virtual network to offer its service. This can be a value-added service and then the SP acts as a application service provider, or a transport service with the SP acting as a network service provider.

These various business roles lead to the architectural entities and organization depicted in Figure 4.1.

In principle a single company can fill multiple roles at the same time. For example it can be PIP and VNP, or VNP and VNO, or even PIP, VNP, and VNO. However, we separate the roles as they are typically performed by distinct groups, even in companies that aggregate multiple roles. For example, running an infrastructure is fundamentally different from negotiating contracts with PIPs about substrate slices. This, in turn, is fundamentally different from operating a specific network, e.g., an IP network for a service provider, which is the task of the VNO. As such, splitting the roles increases our flexibility and facilitates identification of the players, the corporate enterprises, which have a given role. Furthermore, it helps keeping the economic tussles separate from the technical domain.

Note that both a PIP as well as the VNP deliver a virtualized network. Therefore, a VNP can act as a PIP to another VNP. However, one has to keep in mind that a VNP in contrast to a pure PIP has the ability to negotiate contracts with other PIPs and assemble networks.

4.1.1 Player Goals and Tasks

Before we can discuss the interfaces between the roles we investigate the goals and tasks of each individual player in our infrastructure:

Physical Infrastructure Provider: This player seeks to gain a market advantage by maximizing its resource efficiency. It can achieve this, e.g., by over-booking its resources, as well as by optimizing the embedding of VNets to its physical infrastructure. The PIP knows its own topology and can migrate virtual network resources as long as this does not violate any service guarantees. It also knows the respective resource requirements of each virtual subnetwork assigned to it. Its task is the substrate management and thus the realization of the VNets.

VNet Provider: The VNet provider acts as an information broker between PIP offerings and SP requirements. The VNP receives a VNet specification which it may have to transform into multiple sub-VNet specifications, one for each PIP involved. It seeks a competitive advantage by splitting the VNet appropriately and selecting the best value-for-money PIP. It also negotiates the appropriate service level agreements to setup the desired virtual network.

VNet Operator: The VNO provisions and sets up the empty virtual network provided by the VNP. It seeks to minimize its expenses required to operate the VNet. The VNO either receives a VNet specification by the SP or high-level service level requirements. In the latter, case it transforms these higher-level requirements into an appropriate VNet specifications. Generally, we assume that the VNO has no knowledge of how its VNet is realized across one or multiple PIPs. Its task is the day-by-day operation of the VNet.

Service Provider: The service provider is the end-customer of this chain of responsibilities. As such, a SP has to specify service level requirements both for the topology of the virtual network as well as for the operation of the virtual network. Its goal is to receive a network that enables the service to run in an optimal way at low costs.

4.1.2 VNet Application Scenarios

Let us consider some of the new opportunities enabled by our separation of business actors (Figure 4.1) both for existing business entities and new players. Note that players may position themselves anywhere between and PIP and SP. For example, *Player A* can operate as a 'pure PIP', like a bit pipe ISP. On the other side of the spectrum *Player C* may decide to focus on its application service and outsource all other operational aspects. The business entity *Player B* may act as an integrated network manager and provisioner, and offer 'a network as a service' to *Player C* that encompasses VNO and VNP service, buying its bit pipe from *Player A*.

In a different scenario, *Player A* operates as "value-added PIP". It runs its own infrastructures as a PIP, but also acts as VNP that assembles a VNet consisting of parts of its own infrastructure and from other PIPs. *Player B* may then offer the VNO service to the Service Provider *Player C*.

Yet another option is that *Player C* provisions and manages its own VNet in addition to its service, while owning no physical infrastructure. It thus acts as SP, VNO, and VNP, acquiring the basic resources from PIPs A and B. This is similar to the situation in cloud datacenters today that rent out 'infrastructure as a service' to customers who then run and manage their own virtual machines, e.g., Amazon AWS [23] to Reddit [122].

Consider how the proposed GENI architecture [70] fits within our framework. The GENI clearinghouse is a VNP. The experimenter is the VNO and if they desire the SP. As such, GENI also realizes the split between PIP and VNP. However, as GENI does not yet consider federation it does not consider the implications of having to handle multiple PIPs.

Note that in the business context, resource virtualization is already practiced due to high operation costs: Mobile base stations are increasingly shared between mobile operators. Undersea cables are typically not operated by one business alone. Overall, the wholesale business (e.g., sharing of the access network infrastructure) is increasing. As such, VNOs and VNPs already exist even today. But there are currently no well defined interfaces between them.

4.2 VNet Control Architecture

In this section, we introduce our *VNet Control Plane Architecture* which provides the control and management functions for the virtual network architecture to the various actors. The control plane must perform a balancing act between the following tussles:

- Information disclosure against information hiding.
- Centralization of configuration and control against delegation of configuration and control.

The information disclosure tussle is a subtle one and we will try to illustrate this through multiple scenarios. The first scenario is accounting: each customer needs to be able to verify that contractual obligations are met by the provider. Conversely, this must be possible without forcing the provider to release sensitive information to them or others. For example, a customer can request certain Quality of Service (QoS) guarantees across the PIP's network without any information on the exact physical path. The second scenario, which is arguably the most challenging is network debugging. It is a complex problem to provide enough information to, for example, a VNO to enable them to debug a problem with the physical path in PIP without providing too much information.

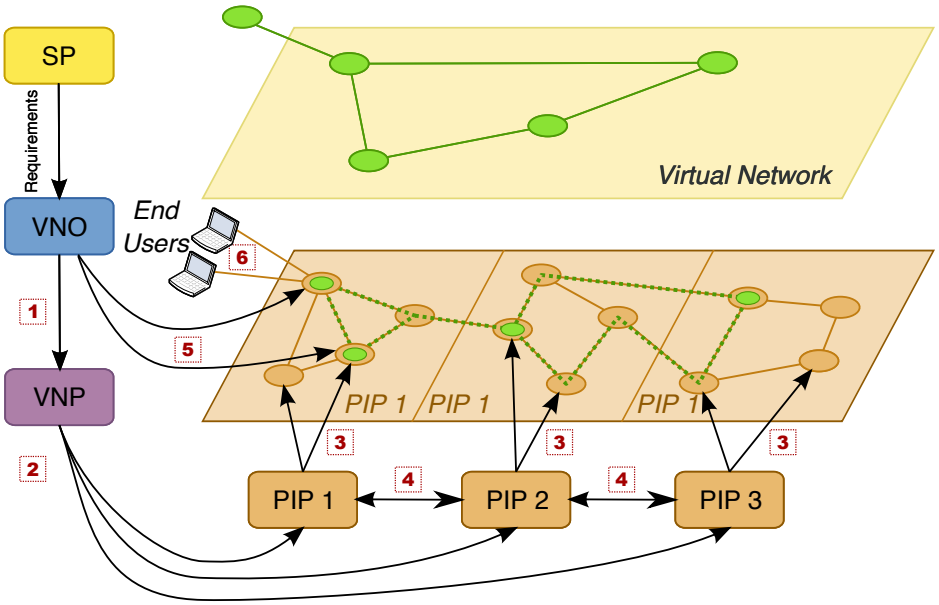


Figure 4.2: VNet control interfaces between players

Where configuration and control are undertaken is another tussle. For example, the PIP should be able to render/delegate low level management of the virtualized network components via the VNP to a VNO, whilst hiding it from another VNO.

4.2.1 Control Interfaces

In the following we identify the control interfaces (see Figure 4.2) in our architecture by discussing how the various players interact in order to setup a VNet.

To begin with, the SP hands the VNO its requirements. Then the VNO adds its requirements and any constraints it imposes on the VNet. This description is subsequently provided (via *Interface 1*) to the VNP of its choice, which is in charge of assembling the VNet. The VNP may split the request among several PIPs, e.g., by using knowledge about their geographic footprints, and send parts of the overall description to the selected PIPs (via *Interface 2*). This negotiation may require multiple steps. Finally, the VNP decides which resources to use from which PIP and instructs the PIPs to set up their part, i.e., virtual nodes and virtual links, of the VNet (*Interface 3*). Now, all parts of the VNet are instantiated within each PIP but they may still have to be interconnected (*Interface 4*). The setup of virtual links between PIPs—in contrast to Interface 3—needs to be standardized in order to allow for interoperability across PIP domains. Once the whole VNet has been assembled the VNO is given access to it (*Interface 5*). This interface is also called "Out-of-VNet" access and is necessary as, at this point in time, the virtual network itself is

not yet in operation. Thus, a management interface outside of the virtual network is needed. Once the virtual network has been fully configured by the VNO and the service is running, end-users can connect to the virtual network (*Interface 6*).

We now discuss how each player benefits from this virtualization architecture: PIPs can better account for the constraints imposed by the VNets. For example, before scheduling maintenance work or for traffic engineering purposes, they might migrate some virtual nodes to minimize downtime or to optimize their traffic flow. This is possible as long as the the new location is embedding-equivalent, i.e., satisfies all of the requirements and imposed constraints, and enabled by the level of indirection introduced by our architecture and the use of modern migration mechanisms [154, 47]. For VNPs, migration between PIPs offers a mechanism to optimize their revenue by choosing competitive and reliable PIPs. As pointed out by [161], the removal of the requirement for individual negotiations between VNOs and all participating PIPs facilitates the entry of new players into the market. Furthermore, SPs may outsource non service specific network operation tasks to other entities and thereby concentrate on their core interests relating to their respective business model. The migration process is transparent to the VNOs. Note that they cannot trigger migration directly; however, by altering their requirements, VNOs may indirectly initiate resource migration.

4.2.2 VNet Instantiation

Setting up a VNet, see Figure 4.3, starts from a VNet specification. For this we need resource description languages for both the VNet topology, including link/node properties, as well as service level requirements. These description languages should neither be too constrained – to allow the VNP and the PIPs freedom for optimizations – nor too vague – to enable a precise specification. We note that the language for specifying service level requirement may be service dependent. However, it should be possible to standardize the VNet topology description language.

To setup the VNet each player, for its domain, has to: formulate resource requirements, discover potential resources and partners, negotiate with this partners based on VNet resource description, and construct the topology. The corresponding communication is depicted in Figure 4.3 as red dotted arrows.

Service Provider: The SP specifies its service specific requirements which might include a VNet topology. In addition, it may specify the kind of interface it needs for service deployment and maintenance, e.g., what level of console access. It then delegates the instantiation to the VNO of its choice. Once the VNet is instantiated the SP deploys its service using the interface provided by the VNO.

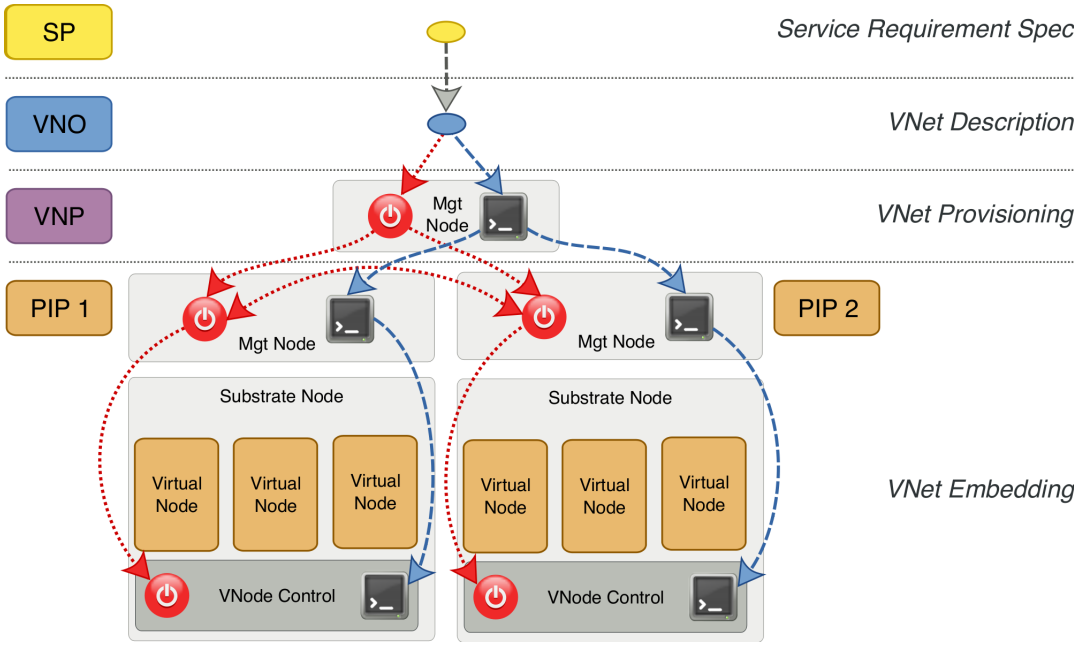


Figure 4.3: Overview of VNet provisioning and Out-of-VNet access.

VNet Operator: The VNO uses the specification it receives from the SP and generates a VNet specification. It then negotiates with various VNPs on the basis of the VNet specification. Once a VNP is selected the VNO waits for the VNP to assemble the VNet. When it has access to the VNet, which consists of a data and a control network it can use the control network, also referred to as Out-Of-VNet access, see Section 4.2.3, to instantiate the network service. Finally, it may instantiate the control interface needed by the SP.

VNet Provider: Upon reception of the VNet resource description, the VNP identifies candidate PIPs and splits the VNet resource description into multiple subsets. It then negotiates with the candidate PIPs regarding the necessary substrate resources. Once the PIPs have assembled the pieces of the VNet, it is completed and connected by the VNP. Finally, the VNP provides a management console access for the whole VNet by relaying the management interfaces of the PIPs. Note, that a VNP may aggregate requests for multiple VNets. It may also request additional resources from the PIPs to satisfy future requests. In this sense, a VNP can act as any reseller would.

Physical Infrastructure Provider: The PIP identifies the appropriate substrate resources, based on the VNet topology descriptions it receives, and allocates them. It has the ability to migrate other VNets in order to free resources for new requests. After setting up the VNet on its substrate it returns both the data and the control part of the VNet. The control part includes the PIP level management consoles to allow the configuration of the virtual nodes. Since VNets may span across multiple PIPs some virtual links may have to be setup across PIPs.

4.2.3 Out-of-VNet Access

Each VNet consists of two logical networks: a data network and a control network. The data network is what one commonly refers to in the context of virtual networks. But, per default, any VNet after instantiation is just an empty set of resources that have to be configured. Moreover, a control interface is necessary during VNet operation for specific VNet management tasks. Such a management access architecture is shown in Figure 4.3 in blue dashed lines. Every player maintains a control interface hosted on dedicated management nodes for "out-of-VNet" access to the VNet resources. This interface provides a console interface as well as control options such as virtual resource power cycling, and also debugging and instrumentation of the Virtual Network.

These communication channels should transparently handle migrations both within the PIP as well as across PIPs. To this end, multiple layers of indirection may need to be used, as indicated in the Figure, e.g., though a chain of proxies.

The different out-of-band interfaces should also be *isolated* from one another and from the data-plane traffic. This enables safe management of the VNet in the presence of data-plane traffic surges, but also protects the production dataplane from the traffic caused by the debugging interfaces, e.g., when tracing or monitoring.

4.2.4 End-user/End-system Access to VNets

End-users/end-systems wishing to dynamically join a VNet need to be authenticated at their physical attachment point in their local PIP before being connected to the VNet. To this end, a dedicated authentication and management VNet can be used. The end-system requests authentication through the authentication VNet. Then, the request is forwarded from its local PIP to the responsible VNO authentication service. Only when authenticated, the end-system is attached to the VNet and configures its address, routing, and other network services inside the VNet.

4.3 Discussion: Benefits and Challenges

We now discuss the benefits and challenges associated with a Virtual Network architecture as outlined above.

Present virtualization approaches: Note that many aspects that comprise a virtual network are already present in today's operator networks. For instance, MPLS is widely employed, and layer-2 VPNs are offered by many providers. However, management of these systems is often semi-manual, and thus complicated and error-prone. Also, no standardized negotiation interfaces exist between providers. As such, these technologies are often only available within a single provider domain. The virtual network architecture presented here standardizes the interfaces, adds the missing cross-provider management interfaces, and thus reduces the need for manual intervention when configuring these features.

Optimize for specific goals: The current Internet has to fulfill many goals at the same time, some of which are outright contradictory. Virtual Network provide the opportunity to optimize each network to a particular goal, e.g., seamless mobility, security, anonymity, low latency, high throughput, cost effectiveness. This can significantly ease the management of each individual VNet. Note that not all VNets have to run the IP protocol stack. It is assumed here that VNet operators can customize their VNet down to the datalink layer. This requires safe access to forwarding control plane each each virtualized device in the VNet.

Service Provider benefits: Service providers like Blizzard, Google and Skype are considered the innovation leaders as well as the most profitable Internet players. These Service Providers could benefit from a virtualized infrastructure by pushing services into the network, and increase the efficiency of existing services, e.g., by imposing latency requirements.

A 'blue pill' [40] for the current Internet: Due to the economical importance of the current Internet, stakeholders may be skeptical to move away from it in one go. Note, however, that the current Internet could continue to operate as one *legacy VNet* in a fully virtualized infrastructure. This avoids having to "change a running system" [65].

Probe-effect free instrumentation and network-wide monitoring: The Out-of-VNet control interface facilitates safe instrumentation and network-wide monitoring of the VNet. By virtue of the isolation, the VNet can still be safely managed and monitored in the presence of traffic surges on the production data plane. Conversely,

the production data plane traffic is also protected from impact by the monitoring traffic, e.g., when large numbers of flows are monitored.

Requirements on the underlying virtualization layer: Many of the benefits of VNets depend on the *isolation* quality provided by the underlying virtualization layers. VNets must not be able to interfere with the safe operation of other VNets. Note that there are different levels of isolation that can be required — in a basic scenario, the virtualization layer provides only *functional isolation*. This inhibits VNets from directly interfering with the operation of other VNets. There may still be *performance interactions* and *sidechannel attacks*. Full performance isolation, in which the performance experienced each individual VNet is completely independent from the behavior of any other VNet is significantly harder to achieve and prevents certain optimizations, e.g., overbooking.

Also depending on the use case, different levels of programmability are required: Custom routing configurations require a partitioned *Forwarding Information Base* (FIB), as well as the ability to run several independent routing processes at the same time. This is the level of programmability offered by the first generation of "virtualizable routers." Custom routing algorithms require the ability to execute custom user processes with access to the *Forwarding Information Base* (FIB). This level of programmability is provided by 2nd generation programmable commercial routers, e.g., via the Juniper JunOS SDK [81]. If custom non-IP protocols stacks are used, VNet operators must be able to run a custom operating system image with a custom protocol instances on their virtual nodes. Today, this is typically possible on virtualized hosts, but not network devices.

Economic tussles: A virtualized infrastructure is sure to create new economic tussles, between the current stakeholders, the *Service Providers* and the (Physical) Infrastructure Operators, over the responsibility for operating and managing VNets. Thus, we require the additional roles defined in our architecture: Virtual Network Providers (VNP) and Virtual Network Operators (VNO).

4.4 Related Work

Over the last years virtual network architectures have been an area of active research. Some groups have focused on using network virtualization to enable larger-scale and more flexible testbeds. Other groups aim at virtualizing production networks or even the Internet. We now revisit the proposals presented in Sections 2.1.3 and 2.1.4, and compare them to to our approach.

Architectures for Experimental Networks: There exist a large number of virtualization architectures for testbeds and experimental networks [36, 150, 38, 53], offering differing degrees of network virtualization. Recall, e.g., that PlanetLab [36] is a distributed, large scale testbed, with a hierarchical model of trust with *Planet Lab Central* (PLC) as its root authority. VINI [150] supports simultaneous experiments with arbitrary network topologies on a shared physical infrastructure and provides rudimentary concepts for end-user attachment [12]. Emulab [53] offers virtual topology configuration based on ns2 configuration files and automatic bootstrapping of experiment nodes.

In GENI [70], a large-scale U.S. initiative for building a federated virtualized testbed, all operations are signed off and managed by a central *Geni Clearing House*, which can thus be regarded as analogue to our VNP. As a possible growth path, GENI plans support for federated clearing houses. During phase 1 of the development both VINI/Planetlab and Emulab are used as GENI prototypes (*ProtoGeni*).

All testbed-oriented architectures mentioned above do not consider several key factors relevant for virtualizing the (commercial) Internet: They assume a hierarchical trust model that centers on a universally trusted entity, e.g., the PLC/GENI clearinghouses. To overcome this limitation, we consider competing players with individual administrative zones that have only limited trust and also have the desire to hide information, e.g., their topologies. Economic models and use cases are not critical for testbed designs but are crucial for the adoption of an Internet-wide virtualization architecture.

Architectures for Production Networks: Prior proposals targeted at production networks have included CABO [63] and Cabernet [161]. CABO proposes to speed up deployment of new protocols by allowing multiple concurrent virtualized networks in parallel with service providers operating their own customized network inside of allocated slices of physical networks managed by infrastructure providers. The idea is refined by Cabernet which introduces a "Connectivity Layer", abstracting the negotiations with different infrastructure providers and allowing for aggregation of several VNets into one set of infrastructure level resource reservations.

While this structure relates to our proposal, our approach differs as we propose to split the service provider and connectivity provider role into the three roles of VNP, VNO, and SP. These roles allow for a more granular splitting of responsibilities with respect to network provisioning, network operation, and service specific operations which may be mapped to different business entities according to various different business models. Furthermore, we extend the framework by considering privacy issues in federate virtual network operations and business aspects.

4.5 Summary

In this chapter, we propose a VNET Control Architecture that comprises four main entities reflecting different business roles: Physical Infrastructure Providers (PIPs), Virtual Network Providers (VNPs), Virtual Network Operators (VNOs), and Service Providers (SPs).

An important aspect of this control architecture is that it defines the relationships that govern the interaction between the players, *without* prescribing their internal organization, structure and policies. In other words, every entity manages its resources as it sees fit. This property is crucial to the viability of the proposal, as it ensures the protection of business interests and competitive advantages. Furthermore, our architecture encompasses other proposed network virtualization architectures, e.g., GENI and Cabernet [70, 161].

In support of this flexible resource management strategy, we emphasize the need for an Out-of-VNet access management interface to allow some basic control of virtual nodes from the outside of the VNet.

5

Safe Evolution and Improved Network Troubleshooting with Mirror VNets

Armed with Virtual Networks as introduced in the last chapter, we now turn our focus from *control* to *troubleshooting*, and investigate how Virtual Networks can improve our troubleshooting abilities. Recall that today diagnosing problems, deploying new services, testing protocol interactions, or validating network configurations are still largely unsolved problems for both enterprise and Internet Service Provider (ISP) networks. Due to the intrinsically distributed nature of network state, frequent timing dependencies, and sources of non-determinism involved, any change may introduce undesired effects—even the impact of a simple configuration change can be hard to predict.

As such, diagnosis is often attempted in "artificial" environments (analytical models, simulators, testbeds) that offer good monitoring capabilities, or in a trial-and-error fashion in the actual production environment. However, these approaches have severe limitations: Models and simulations have limited prediction capabilities due to modeling assumptions and abstractions, testbeds are limited in scale due to cost factors, and trying to fix things on the production network often leads to other errors [62, 97]. Therefore, all these approaches often do not suffice to diagnose and then resolve real-life network problems since these often stem from complex interactions of many parties. Problems that only occur sporadically or are triggered by user interactions are known to be especially problematic, even if it is possible to replicate a complete setup in a lab environment.

We propose to utilize Virtual Networks (VNets) to help tackle these network diagnosis and troubleshooting problems. Recall that VNets expand the existing concepts of virtualized hosts and links to the entire network. Therefore, a VNet may span multiple physical network domains. VNet management frameworks, such as our control architecture presented in the last chapter, unify the management of these resources and enable VNets to be dynamically provisioned and configured and to operate in parallel on a shared physical infrastructure. A controlling instance *isolates* the individual VNets from each other. As a result, it is possible to experiment with one VNet, while maintaining stability in the rest of the system.

In this chapter we show how properly implemented VNets enable us with good diagnosis and debugging capabilities. We propose *Mirror VNets*, which replicate networks and traffic in a safe fashion. Thus, the new Mirror VNet and the production VNet can operate in parallel and the user traffic is duplicated either completely or in part to both networks.

This allows the network operator to investigate a problem and locate its root cause in the Mirror VNet without interfering with the production traffic. Then, a fix can be developed, tested, and deployed even with real user traffic again, without affecting the production setup. Once the network operator is convinced that the new setup is stable and fixes the problem, he can switch the production VNet with the Mirror VNet in a near-atomic fashion. Therefore, Mirror VNets offer network operators a new range of capabilities from (a) safely evaluating and testing new configurations, via (b) testing new software components at same scale of the production network and under real user traffic, to (c) troubleshooting live networks.

This approach builds on the agility and isolation properties of the underlying virtualized infrastructure and does not require changes to the physical or logical structure of the production network. Instead, a network owner can dynamically provision a Mirror VNet as required. It does not require changes to the production network within the VNet— its protocols, its applications, or its configuration.

This chapter makes the following contributions:

(i) we introduce *Mirror VNets*

(ii) discuss their intrinsic benefits and limitations,

(iii) present an implementation based on XEN and OpenFlow,

(iv) a practical case study in a multimedia / QoS context

(v) and evaluate the mirroring performance of the Linux kernel.

5.1 Mirror VNets

In this section, we show how network virtualization enables Mirror VNets. Then we discuss Mirror VNet use-cases and discuss their intrinsic benefits and limitations.

5.1.1 Assumptions

Our approach is based on the assumption that the network is virtualized. This includes node as well as link virtualization with proper isolation and resource management and accountability. As mentioned, many virtualized resources are present even in today's network infrastructures, e.g., VLANs, MPLS, VPNs, and comprehensive management frameworks are currently emerging. Virtualization is also considered a key enabler for the Future Internet [64]. In addition, we assume that the infrastructure is shared among a reasonable number of VNets and thus none of the VNets uses more than a fraction, say 10%, of the overall substrate resources.

Another assumption is that we can duplicate input traffic to multiple VNets. Such capability is also commonly deployed in today's network infrastructure, e.g., spanning ports on Ethernet switches, multicast in routers, optical splitters, lawful intercept, and the functionality offered by OpenFlow.

5.1.2 Approach

To upgrade or troubleshoot a production VNet (VNet A), e.g., in Figure 5.1, the operator clones it and pairs it with a parallel Mirror VNet (VNet B). Thus VNet B starts with the identical configuration and state as VNet A, e.g., by relying on the same techniques as used for network node migration. The operator ensures that the input traffic for VNet A is *mirrored* either completely or in part to VNet B. This has to happen at all attachment points between the VNet and external entities, e.g., end-systems or network entry points. All user traffic traverses both VNets, however, only traffic from the production VNet A is send back to external nodes. Traffic from the Mirror VNet B is discarded, silently.

Both VNets A and B now operate in parallel. Moreover, they are fully isolated from each other, on a node- and link-level. Thus, from now on they operate independently but under the same (or partial) user traffic. This means that it is now possible to use VNet B to troubleshoot network problems, test a new software or hardware release or reconfigure the network.

Once the problem has been diagnosed and a solution has been found, deployed to the Mirror VNet B and validated, the Mirror VNet B can be upgraded to become the new production VNet. To this end, the VNet attachment points of both VNets to external hosts are swapped: External hosts now communicate with VNet B, and

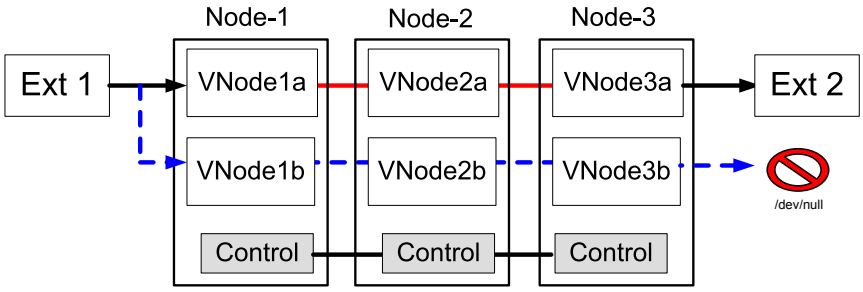


Figure 5.1: Mirror VNet example: substrate running production VNet A, consisting of VNodes $1a, 2a, 3a$, and Mirror VNet B, consisting of VNodes $1b, 2b, 3b$

VNet A acts as a Mirror VNet, and its output is discarded. This is a relatively simple operation and can be completed quickly. Finally, the old VNet A can be dismantled and its substrate resources released.

Note that not all traffic has to be mirrored and not all mirrored traffic needs to be transmitted over the wire. Depending on the scenario, the operator can use *mirror strategies* to adapt the volume of the traffic to be mirrored and transmitted. Possible Mirror strategies include:

(a) *Full Mirror:* Every packet is mirrored verbatim and transmitted independently in both VNets.

(b) *Packet Headers:* Only packet headers are transmitted in the Mirror VNet, the payload is reconstructed with dummy data at the nodes.

(c) *Traffic Stats:* Instead of actual packets, only aggregated traffic statistics are transmitted and similar traffic is reconstructed at the nodes under investigation.

(d) *Sampling:* Only a selected fraction of packets or flows is mirrored to the Mirror VNet.

These strategies can dynamically and selectively be applied to parts of the traffic. Note, for instance, that Control plane traffic has been shown to account for $< 1\%$ of traffic volume, but cause $> 95\%$ of bugs [22]. Accordingly, many problems (e.g., routing problems) can be investigated by only mirroring control-plane messages verbatim. For the data-plane traffic, headers or even reconstructed summaries may well suffice. Stress testing of a new software release can be started at only a fraction, e.g., 5%, of the overall traffic.

5.1.3 Use-cases

Mirror VNets can be useful in many scenarios, including routing optimizations and updates of network services.

Routing configuration optimization: When a network operator decides that he wants to re-optimize routing, e.g., by changing the link weights of his interior routing protocol or by introducing routing policies, he often does not want to experiment on the production network. Among the drawbacks are unintentional path choices, link overloads, packet reordering and losses during convergence, erroneous configurations, etc. However, by using a Mirror VNet one can perform all changes while traffic in the production VNet continues to flow unaffected. Indeed, the operator can monitor the Mirror VNET for the expected benefits or unexpected side effects. Only when the operator is "happy" with the new configuration state, he switches roles, and exposes the new routing to his users. The capacity overhead is small as it is sufficient to mirror routing protocol messages exchanged between external entities and the Mirror VNet.

Update of a faulty network service: Consider a wide area network service based on an overlay network, e.g., a popular Internet telephony system. Suppose a number of overlay supernodes crash in irregular intervals, due a regression bug [14]. In this case, the operator can instantiate a Mirror VNet, in the exact same state as the production VNet. The bugs responsible for the problem are tracked down by manipulating selective traffic passed to the Mirror VNet. When a potential fix has been developed, the new version is deployed to the Mirror VNet, and an increasing amount of traffic is mirrored to the Mirror VNet to validate the bugfix, and check against new regressions that may have been introduced.

This can introduce some overhead especially with regards to bandwidth capacity. However, the benefit is that the new software components are stress-tested under real user traffic and are effectively probed for possible unknown regressions, some of which may be very hard to discover in analytical models or test-labs.

When the new version proves stable under full input traffic, and thus sufficient confidence has been gained, the outputs of Mirror and production VNets are switched.

5.1.4 Discussion

The benefits of Mirror VNets are many-fold and include:

Resilience against operational mistakes: Mistakes during the configuration and/or update process are limited to the Mirror VNet. Thus they do not affect the production network. Moreover, the entire change set is tested under realistic conditions before affecting production.

Real user traffic: Mirror VNets expose the new system and its configuration to real user traffic at full scale and thus offer the opportunities to detect more bugs earlier.

Complex setups: Some problems can only be reproduced in complex setups, which can not be reproduced with models, simulations, or in test-labs. The only way to test under real-world conditions is to deploy in a real network – however, mirror VNets enable us to do this in a safe manner, without impacting the reliability of the production network.

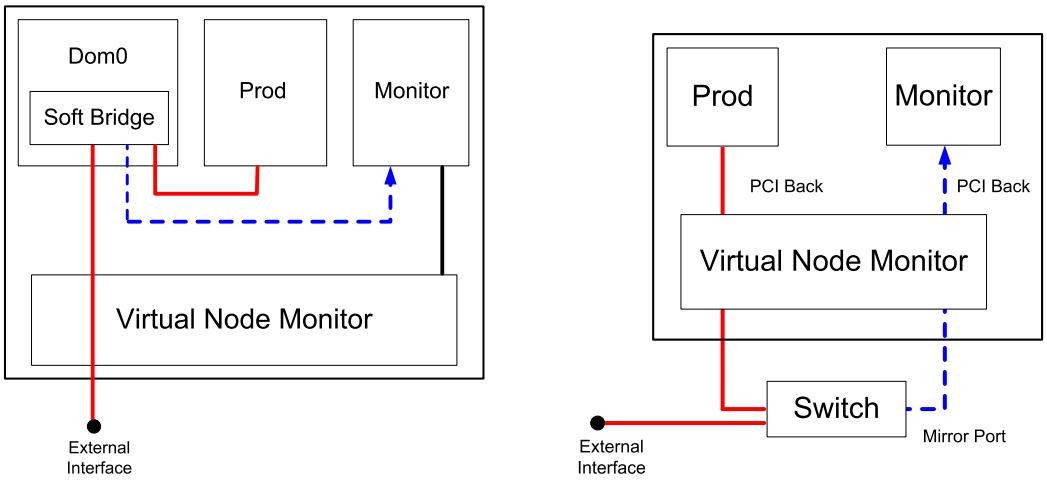
Rollback/undo for networks: If after being set into production, an upgraded Mirror VNet exposes unexpected problematic side effects (e.g., due to a closed loop effect, see below), the operator can easily revert back to the old production network, and continue to investigate the problem without affecting the production users any longer.

An inherent limitation of this approach is that closed-loop effects caused by hosts outside the VNets cannot be predicted by monitoring the Mirror VNet. For example, an upgrade that provides faster delivery of requests to an external server may result in faster response traffic which may cause an overload on the network. Such effects only occur once the Mirror VNet is made productive and thus cannot be predicted by monitoring the Mirror VNet while it is still in mirror mode. Note that closed-loops *within* the VNets are not affected by this problem, so one possible solution is to integrate all communicating parties, including the end hosts, *into* the VNet itself, and have the VNet operate *end-to-end*.

Another concern is that the substrate must carry the additional load imposed by the Mirror and the Control VNets. Note that depending on the problem under investigation, the mirroring strategies discussed in Section 5.1.2 can reduce the amount of traffic that actually has to be mirrored significantly, and stress-testing can be done gradually (e.g., at 10% of the traffic load). Finally, assuming many co-existent VNets on a well dimensioned (e.g., 2/3 loaded) substrate, debugging a limited number even at Full-Mirror mode is feasible. If the impact of a proposed change cannot be determined by other means, then the capacity-overhead may reflect a fair price. In the event of an unexpected traffic or load spike, the production VNets are always granted priority over Mirror VNets. Therefore, debugging or testing is affected but the user service in the production VNet is not deteriorated.

5.2 Prototype Implementation

This work assumes the presence of a virtualization solution that enables duplication of a virtual machine. In addition, we require the network substrate to be able to dynamically replicate traffic as required. For our case study and performance evaluation, we build a prototype Mirror VNets system.



(a) internal bridging (b) external bridging

Figure 5.2: Mirror VNets: Options for link attachment

From the host virtualization options discussed in Section 2.1.2 we choose XEN. XEN has been shown to be a viable solution for building high performance routers on commodity hardware [59] if functional blocks are selected and placed carefully. Performance isolation and fairness issues can be challenging especially with regards to network I/O [38, 57]. Highly beneficial for our case, XEN also supports *live migration*. Based on this feature, we can extend the hypervisor to support instant creation of mirror nodes in identical state. This essentially corresponds to a live-migration to `localhost`, without disbanding the original guest. Our prototype system is based on XEN 3.4.

On the link layer, we consider several options for link attachment to study the performance and isolation by current virtualization techniques. Option (a) uses the standard Linux software bridge, see Figure 5.2(a). Option (b) maps the NIC directly into the DomUs and thus hides it from the Dom0 (`pciback` see Figure5.2(b)).

In Option (a) we duplicate the packets inside the host, using `tc`. In Option (b) this is not possible. Packets have to be inspected and duplicated externally. We can use a statically configured switch port or a dynamically configured open flow switch for this purpose. In this work, we simulate the use of Multi-Queue NIC cards by using several dedicated cards, and use an OpenFlow-enabled switch for traffic duplication. As introduced in Section 2.2, OpenFlow [104] enables an external entity, the controller, to control Ethernet switches. This controller can dynamically set the forwarding rules using wildcard patterns across the packet headers while the frame forwarding is done by the switch hardware. Thus, OpenFlow is a flexible and powerful solution for the link substrate capabilities required by our approach.

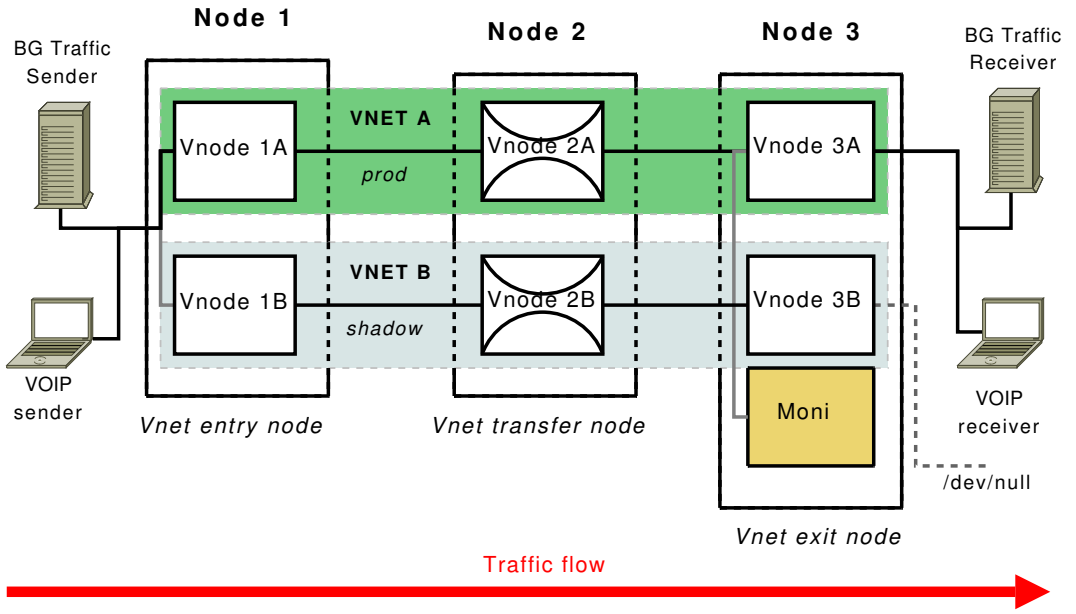


Figure 5.3: Mirror VNet experiment setup

Our prototype implementation uses OpenFlow 0.8.9, the current version at the time of the study. A custom OF controller based on NOX serves the purpose of mirroring the incoming traffic to both VNets and filtering outgoing traffic according to the status.

5.3 Case Study

We present a case study highlighting the potential of Mirror VNets for troubleshooting a problem in a production network. Consider the following scenario: A VNet operator that offers both VoIP and Internet access across a best effort VNet, considers moving to a setup with service differentiation to offer better quality of service (QoS) to its VoIP traffic.

For our experiment, a VoIP call and background traffic of varying intensity is routed through the virtualized substrate, shown in Figure 5.3). The substrate network consists of three nodes. We now instantiate two parallel VNets, VNet A and B, each with a maximum bandwidth of 20 Mbit/s throughout the experiment, enforced by traffic shaping on node 2. Moreover, we setup an additional virtual network for monitoring. On entry to the VNet, traffic is duplicated to both VNets A and B and forwarded within each via node 2 to node 3 using separate virtual links (VLANs). On exit, when leaving node 3, only output from one VNet is sent to the receivers. The monitoring VNet "Moni" receives a copy of the VoIP traffic from both VNets.

Figure 5.4: Overview of MoS voice quality ratings

5.3.1 Experiment Metrics

For our evaluation, we measure at two points in the experiment: Moni records data for both VNets on exit of the VNet, while the receiver records the quality as experienced by the user. We record the percentage of dropped packages on the VoIP call as a rough quality indicator, and calculate the *Mean Opinion Score* (MoS) as defined by the ITU-T E-model [60][1].

5.3.2 Experiment Outline

For the VoIP traffic we use the *pjsip* [117] client, an open source VoIP client based on SIP. It generates traffic at a constant rate of 80 kb/s using the $G.711$ codec with a net bitrate of 64 kb/s. Each VoIP RTP packet contains 20ms voice and has a payload of 160 Bytes. A pool of servers is used to generate the background traffic, using Harpoon [138], with properties that are consistent with those observed in the Internet – heavy-tailed file distributions and self-similar traffic, emulating the Internet access traffic by the users of the VNet. To account for different intensities of the background traffic during different times of the day we use two different load levels: L/H that correspond to 20-25%, 60-86% average link utilization. All traffic sources are located on the left in Figure 5.3.

The experiment is conducted in six phases with a length of five minutes each. Figure 5.5 shows the rate of the background traffic (shaded area) averaged over 10s (scale

[1]The E-model states that the various impairments contributing to the overall perception of voice quality (e.g., drops, delay, jitter) are additive when converted to the appropriate psycho-acoustic scale (*R factor*). The *R-factor* is then translated via a non-linear mapping to the *Mean opinion Score*(MoS), a quality metric for voice. MoS values range from 1.0 (*not recommended*) to 5.0 (*very satisfied*). Voice quality classes along with their respective R factor and MoS values are shown in Figure 5.4.

Phase	1	2	3	4	5	6
Active VNets	A	A	A&B	A&B	A&B	B
Production VNet	A	A	A	A	B	B
QoS enabled	-	-	-	B	B	B
BG traffic load	L	L/H	H	H	H	H

Table 5.1: QoE case study: experiment outline across phases

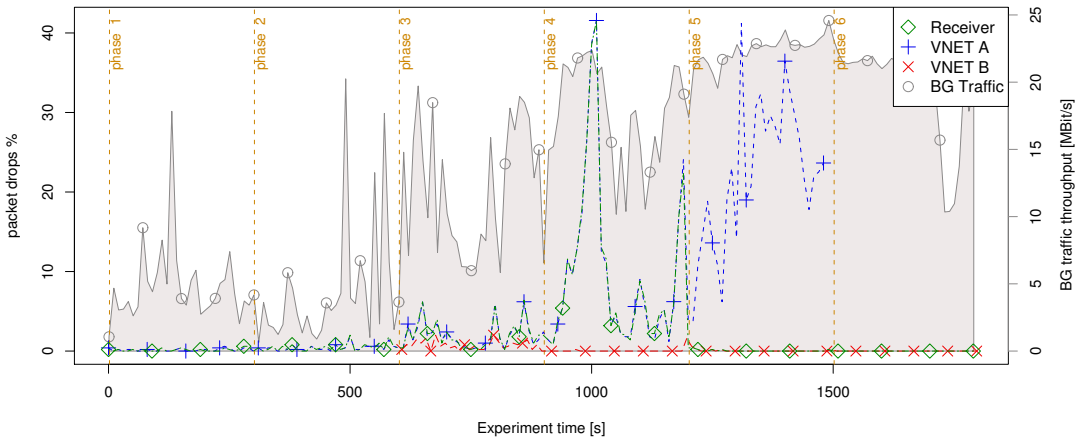


Figure 5.5: Timeseries of VoIP packet drops (left) and background traffic throughput (right) during the experiment, as measured within VNets A, B and at the end host (receiver)

on right axis) across time. In addition, Figure 5.5 shows the number of dropped packets across time, again using 10s bins (scale on left axis). Drop rates for VNet A are depicted as blue plus signs, VNet B as red crosses, and the values measured at the receiver as green diamonds. Table 5.1 summarizes the configuration of each phase.

5.3.3 Results

In phase 1, background traffic is running at low intensity. In the middle of phase 2 the intensity of the Internet traffic is switched to high. This causes a problem in VoIP quality as measured by the MoS value, see Figure 5.6. The perceived quality drops from a MoS score of 4.34 which corresponds to a "very satisfied" service level drops to 4.16 which corresponds to a level of "satisfied".

As such, the VNet operator asks to instantiate a Mirror VNet at the beginning of phase 3. This means that all packets are now duplicated at node 1 and are routed in both VNets A and B. However, the end user for VoIP service is still getting service through VNet A. This allows the operator to assess the impact of the degradation and to do root cause analysis in VNet B. Indeed, the quality of the call decreases

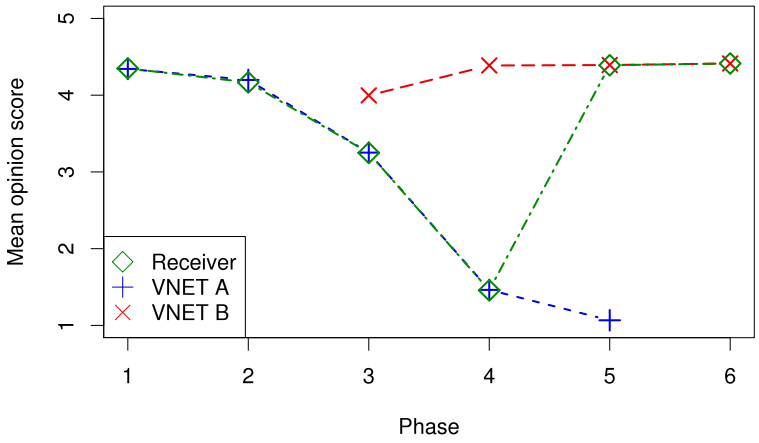


Figure 5.6: Mean MoS value per experiment phase, as measured within VNets A, B and at the end host (receiver).

further in our experiment. In our case the operator decides to prioritize VoIP traffic to counter the bad performance. He enables QoS at the start of phase 4. Even though the background traffic increases further in this phase, the QoS reduces the loss rates within VNet B significantly and the MoS value increases again to 4.38. At the same time, VNet A is hit hard by the increased traffic, causing heavy congestion. In consequence, the VoIP MoS score drops to 1.45 ("not recommended").

At the start of phase 5 the operator switches his production VNet from VNet A to VNet B. Note that packet drops as experienced by the end user do not increase noticeably during the switch. After the switch is completed, the user can profit from the good performance provided by VNet B. With phase 6 the operator deactivates VNet A.

This very simple scenario shows how an operator can benefit from Mirror VNets, e.g., to smoothly upgrade his network configuration to amend a network performance problem. Early results indicate that the approach also works for the other use cases mentioned in Section 5.1.3 and that it can scale to larger topology sizes. Ongoing evaluation within larger OpenFlow-enabled infrastructures that are currently being deployed will provide additional insights into the scalability.

5.4 Mirroring Performance

To assess the feasibility of network-wide Mirror VNets, we need to study the performance and isolation provided by current virtualization approaches. We now study the mirroring performance utilizing XEN for host virtualization and VLANs for link virtualization. To study the limits of the isolation offered by current virtualization techniques, we compare multiple options of link attachment and packet duplication.

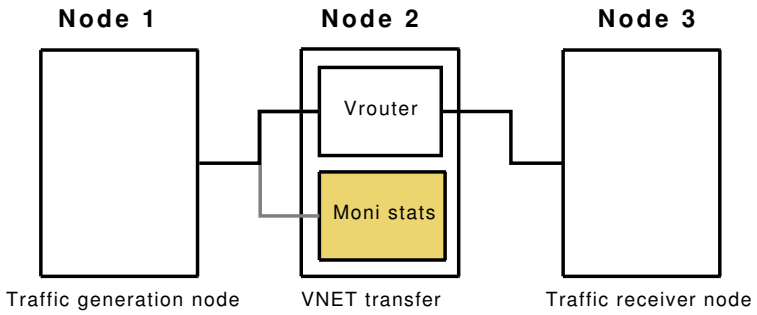


Figure 5.7: Mirroring performance experiment setup

5.4.1 Evaluation Setup

For our evaluation we rely on a three node setup. Each node has two Quad Core Intel Xeon L5420 processors running at 2.5GHz, 16GB of RAM, and 4-8 1GBit/s Intel Ethernet ports. The schematic setup is shown in Figure 5.7. We deploy Debian Linux 4.0 with XEN 3.0.3, which is part of the distribution. We use this out-of-the-box configuration as it resembles a possible production deployment better than custom, hand optimized kernel and hypervisor versions. The virtual network consists of 3 nodes. Node 1 is the traffic source while Node 3 is the sink. Node 2 forwards the traffic from Node 1 to Node 3.

5.4.2 Forwarding Results

On UNIX systems forwarding performance is usually dominated by the per-packet overhead. As such, our baseline experiment uses minimum sized packets and explores the performance impact of the various options on how to attach the link to the virtual node and how to do packet duplication, see Section 5.2. Figure 5.8 shows the boxplots of the forwarding performance within 1 second time bins for an experiment duration of 500 seconds. Results are grouped according to the phases of the experiment by the dashed vertical (red) lines.

In the first two phases, we baseline our setup by measuring the forwarding rate in native Linux (group 1) and Dom0 (group 2). Native Linux (Debian 2.6.18-6) supports a median forwarding rate of 840 kpps. Interestingly, the XEN kernel performs better when doing native, unbridged forwarding in Dom0 (939.9 kpps). Next, we introduce the soft-bridge that is required for internal attachment of the DomUs, but still keep the forwarding in Dom0. We notice that the performance drops to 522 kpps. We then switch to option (a) by delegating the forwarding to DomU via a soft-bridge (third phase). Notice that the forwarding performance drops to 215 kpps even without monitoring, After enabling mirroring forwarding performance is further reduced to

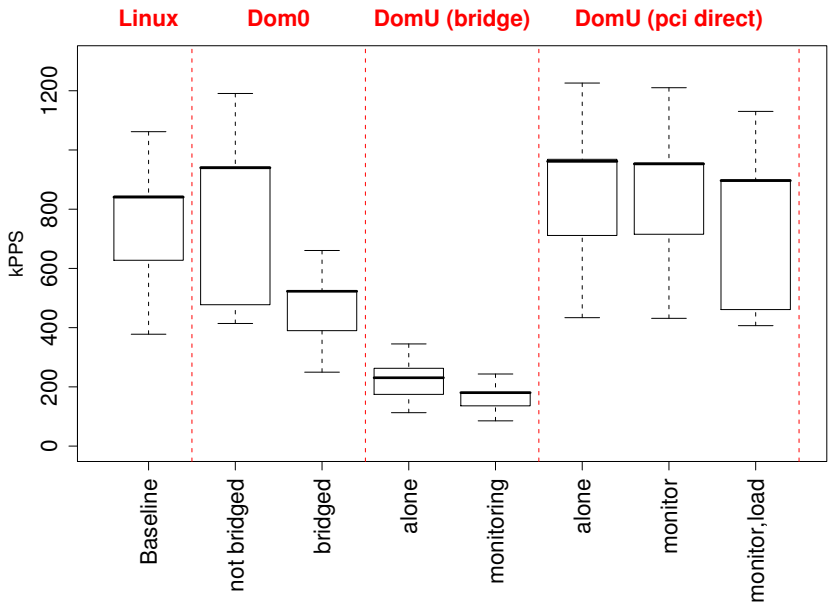


Figure 5.8: Boxplot of forwarded packets packets/second for 64-byte packets, with and without Mirroring

180 kpps which indicates that network performance isolation is a problem in this setup.

Next we study option (b) (directly attaching the DomU interface to the NIC via `pcidirect`, group 4 in the figure). There is hardly any performance degradation. Indeed, we see a performance improvement: without monitoring, a directly attached DomU forwards at 961.6 kpps. We then enable packet duplication via option (b) (simulated OpenFlow packet duplication using a manually configured switch monitoring port). As expected, the performance impact is minimal – forwarding is at 953 kpps. Even when the monitoring domain is overloaded, with 100% CPU load and 100% hard drive I/O load, forwarding performance degrades by only 5% to 896 kpps.

We conclude that probe-effect free Mirror VNets are possible. However, their performance strongly depends on the isolation properties offered by the virtualization platform for network I/O.

Naive soft-bridge approaches result in severe performance penalties, but we were able to demonstrate that direct PCI attachment and external mirroring removes this bottleneck, and that with the deployment of emerging technologies like OpenFlow, large-scale, probe effect-free Monitoring VNets will indeed be feasible.

5.5 Related Work

Our work bears some similarities to the *Shadow Config* approach by Alimi et al. [19], designed to improve the safety and smoothness of configuration updates, but by virtue of the underlying virtualization extends the scope beyond configuration changes. Another sibling to our approach has been proposed by Lin et al. [93]. The authors introduce a framework that enables full reproducibility by recording and replaying all non-deterministic events. Note that the necessary coordination and transaction modules inside the Hypervisor add significant complexity to the *production datapath* and thus cannot be fully transparent to the production network. Numerous other approaches for improving troubleshooting support in networks have been proposed. Some authors [26, 67] propose adding pervasive tracing support throughout the network, which is difficult because it requires changes to all involved hosts, routers, and software stacks.

Active probing has been proposed for fault localization at the network layer. For example, Badabing [139] and Tulip [96] measure per-path characteristics including loss rate and latency, to identify problems that impact perceived performance. Automatic inference [16, 32, 125] can be used to detect problems, profile applications and in some cases infer root causes from traces collected by network monitoring.

Gupta [78] with *DieCast* suggest to scale down physically large-scale distributed systems and map them to smaller virtualized testbeds. They combine *Time Dilation* [79], disk I/O simulation, and of-the-shelf system virtualization technologies, e.g., XEN [35] to run the system at slower pace with reduced hardware resources. Note that sporadic problems or problems induced by unforeseen user behavior remain challenging to reproduce in a testbed.

A complementary approach [87] is to use virtualization to improve bug tolerance in software routers. They use a virtualized environment to run differing implementations and versions of routing software. They use a specialized proxy to distribute incoming routing message to all these routing daemons. Outgoing messages are decided up the routing proxy by applying a weighted voting scheme.

5.6 Summary and Future Work

In this chapter, we present an approach that adds network troubleshooting capabilities to virtualized enterprise or ISP networks. *Mirror VNets* enable operators to upgrade configurations and software in an operationally safe manner with transaction semantics while exposing the new system and configuration to real user behavior. Less expensive, specific problems can be investigated and debugged, and interactions can be studied by sending only selected traffic to the Mirror VNet, e.g., control plane

messages. The experiences with our prototype implementation underline the feasibility of the approaches, especially if used on a virtualization platform that offers good isolation. In the future, we plan to further evaluate the scalability of the system in the large scale OpenFlow-enabled testbeds currently planned, e.g., the Ofelia testbed [4]. The capabilities of Mirror VNets for safe evolution and updates, as well as troubleshooting are of particular interest in such environments where real users are using experimental software.

6

OFRewind: Enabling Record and Replay Troubleshooting for Networks

Lastly, we now turn to the far end of the communication path, the server-side edge of the Internet. Increasingly, services as our online web-based spreadsheet example are provided in large cloud datacenters, as operators seek to exploit economies of scale. In many cases these data-centers are virtualized and multi-tenant, and accommodate many customers at the same time.

Troubleshooting such operational networks can be a daunting task, due to their size, distributed state, and the presence of *black box* components such as commercial routers and switches, which are poorly instrumentable and only coarsely configurable. The tool set available to administrators is limited, and provides only aggregated statistics (SNMP), sampled data (NetFlow/sFlow), or local measurements on single hosts (tcpdump).

In this chapter, we leverage *split forwarding architectures* such as OpenFlow to add *record and replay debugging* capabilities to networks – a powerful, yet currently lacking approach. We present the design of **OFRewind**, which enables *scalable*, *multi-granularity*, *temporally consistent recording* and *coordinated replay* in a network, with fine-grained, dynamic, centrally orchestrated control over both record and replay. Thus, **OFRewind** helps operators to reproduce software errors, identify data-path limitations, or locate configuration errors.

Replay can be performed over parts of the control or data traffic, over alternate hardware or ports, and at a precise time pace. We demonstrate its use se in several case studies from a production OpenFlow network and evaluate its scalability.

Motivating Anecdote

Consider the following anecdotal evidence, showcasing how problem localization and troubleshooting in operational networks remain largely unsolved problems today.

Towards the end of October 2009, the administrators of the Stanford production OpenFlow network began observing strange CPU usage patterns in their switches. The CPU utilization oscillated between 25% and 100% roughly every 30 minutes and led to prolonged flow setup times, which where unacceptable for many users. The network operators began debugging the problem using standard tools and data sets, including SNMP statistics, however the cause for the oscillation of the switch CPU remained inexplicable. Even an analysis of the entire control channel data could not shed light on the cause of the problem, as no observed parameter (number of: packets in, packets out, flow modifications, flow expirations, statistics requests, and statistics replies) seemed to correlate with the CPU utilization. This left the network operator puzzled regarding the cause of the problem.

This anecdote (further discussion in Section 6.3.2) hints at some of the challenges encountered when debugging problems in networks. Networks typically contain *black box* devices, e.g., commercial routers, switches, and middleboxes, that can be only coarsely configured and instrumented, via command-line or simple protocols such as SNMP. Often, the behavior of black box components in the network cannot be understood by analytical means alone – controlled replay and experimentation is needed.

Furthermore, network operators remain stuck with a fairly simplistic arsenal of tools. Many operators record statistics via NetFlow or sFlow [116]. These tools are valuable for observing general traffic trends, but often too coarse to pinpoint the origin problems. Collecting full packet traces, e.g., by tcpdump or specialized hardware, is often unscalable due to high volume data plane traffic. Even when there is a packet trace available, it typically only contains the traffic of a single VLAN or switch port. It is thus difficult to infer temporal or causal relationships between messages exchanged between multiple ports or devices.

Previous attempts have not significantly improved the situation. Tracing frameworks such as XTrace [67] and Netreplay [26] enhance debugging capabilities by pervasively instrumenting the entire network ecosystem, but face serious deployment hurdles due to the scale of changes involved. There are powerful tools available in the context of distributed applications that enable fully deterministic recording and replay, oriented toward end hosts [69, 89]. However, overhead for the fully-deterministic recording of a large network with high data rates can be prohibitive and the instrumentation of 'black' middleboxes and closed source software often remains out of reach.

Introducing OFRewind

In this chapter, we present a new approach to enable practical network recording and replay, based upon an emerging class of network architectures called *split forwarding architectures* as introduced in Section 2.2, e.g., OpenFlow [104], Tesseract [158], and Forces [5]. Recall that these architectures *split* control plane decision-making off from data plane forwarding. In doing so, they enable custom programmability and centralization of the control plane, while allowing for commodity high-throughput, high-fanout data plane forwarding elements.

We discuss, in Section 6.1, the design of **OFRewind**, a tool that takes advantage of these properties to significantly improve the state-of-the-art for recording and replaying network domains. **OFRewind** enables *scalable*, *temporally consistent*, *centrally controlled* network recording and *coordinated* replay of traffic in an OpenFlow controller domain. It takes advantage of the flexibility afforded by the programmable control plane, to dynamically *select* data plane traffic for recording. This improves data-path component scalability and enables *always-on* recording of critical, low-volume traffic, e.g., routing control messages. Indeed, a recent study has shown that the control plane traffic accounts for less than 1% of the data volume, but $95 - 99\%$ of the observed bugs [22]. Data plane traffic can be *load-balanced* across multiple *data plane recorders*. This enables recording even in environments with high data rates. Finally, thanks to the centralized perspective of the controller, **OFRewind** can record a *temporally consistent* trace of the controller domain. This facilitates investigation of the temporal and causal interdependencies of the messages exchanged between the devices in the controller domain.

During replay, **OFRewind** enables the operator to select which parts of the traces are to be replayed and how they should be mapped to the replay environment. By partitioning (or *bisecting*) the trace and automatically repeating the experiment, our tool helps to narrow down and isolate the problem causing component or traffic. A concrete implementation of the tool based on OpenFlow is presented in Section 6.2 and is released as free and open source software [8].

Our work is primarily motivated by operational issues in the OpenFlow-enabled production network at Stanford University. Accordingly, we discuss several case studies where our system has proven useful, including: switch CPU inflation, broadcast storms, anomalous forwarding, NOX packet parsing errors, and other invalid controller actions (Section 6.3). We in addition present a case study in which **OFRewind** successfully pinpoints faulty behavior in the Quagga RIP software routing daemon. This indicates that **OFRewind** is not limited to locating OpenFlow-specific bugs alone, but can also be used to reproduce other network anomalies.

Our evaluation (Section 6.4) shows *(a)* that the tool scales at least as well as current OpenFlow hardware implementations, *(b)* that recording does not impose an undue

performance penalty on the throughput achieved, and *(c)* that the messaging overhead for synchronization in our production network is limited to 1.13% of all data plane traffic.

Key Observations

While using our tool, we have made and incorporated the following key observations:

I. A full recording of all events in an entire production network is infeasible, due to the data volumes involved and their asynchronous nature. However, one usually needs not record all information to be able to reproduce or pinpoint a failure. It suffices to focus on *relevant subparts*, e.g., control messages or packet headers. By selectively recording critical traffic subsets, we can afford to turn *recording on by default* and thus reproduce many unforeseen problems *post facto*.

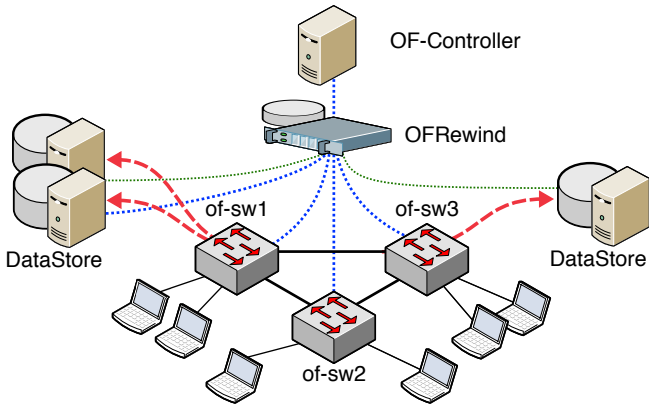
II. Partial recordings, while missing some data necessary for fully deterministic replay, can be used to reproduce symptomatic network behavior, useful for gaining insights in many debugging situations. With careful initialization, the behavior of many network devices turns out to be *deterministic with respect to the network input*.

III. By replaying *subsets of traffic* at a *controlled pace*, we can, in many cases, rapidly repeat experiments with different settings (parameters/code hooks) while still reproducing the error. We can, for instance, *bisect* the traffic and thus localize the sequence of messages leading to an error.

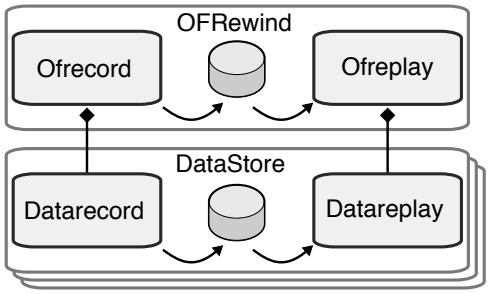
In summary, **OFRewind** is, to the best of our knowledge, the first tool which leverages the properties of split architecture forwarding to enable practical and economically feasible recording and replay debugging of network domains. It has proven useful in a variety of practical case studies, and our evaluation shows **OFRewind** does not significantly affect the scalability of OpenFlow controller domains and does not introduce undue overhead.

6.1 OFRewind System Design

In this section, we discuss the expected operational environment of **OFRewind**, its design goals, and the components and their interaction. We then focus on the need to synchronize specific system components during operation.



(a) System components



(b) Software modules

Figure 6.1: Overview of **OFRewind**

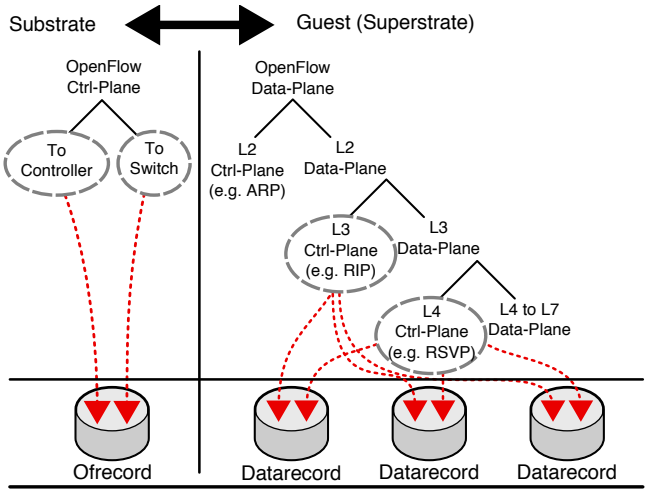


Figure 6.2: OFRewind Traffic strata

6.1.1 Environment / Abstractions

We base our system design upon *split forwarding architectures*, for instance, OpenFlow [104], Tesseract [158], or Forces [5], in which *standardized data plane elements* (switches) perform fast and efficient packet forwarding, and the control plane is *programmable* via an external controlling entity, known as the *controller*. Programmability is achieved through *forwarding rules* that *match* incoming traffic and associate them with *actions*. We call this layer of the network the *substrate*, and the higher-layer *superstrate* network running on top of it *guest*. We call the traffic exchanged between the switches and the controller the substrate *control plane*. The higher-layer control plane traffic running inside of the substrate *data plane* (e.g., IP routing messages) is called the guest *control plane*. The relationship between these layers and traffic strata is shown in Figure 6.2.

Even though not strictly mandated almost all split-architecture deployments group several switches to be centrally managed by one controller, creating a *controller domain*. We envision one instance of **OFRewind** to run in one such controller domain. Imagine, e.g., a campus building infrastructure with 5-10 switches, 200 hosts attached on Gigabit links, a few routers, and an uplink of 10GB/s.

6.1.2 Design Goals and Non-goals

As previously stated, recording and replay functionalities are not usually available in networks. We aim to build a tool that leverages the flexibility afforded by split-architectures to realize such a system. We do not envision **OFRewind** to do automated *root-cause analysis*. We do intend it to help *localize* problem causes. Additionally, we do not envision it to *automatically tune recording* parameters. This is left to an informed administrator who knows what scenario is being debugged.

Recording goals: We want a *scalable* system that can be used in a realistic-sized controller domain. We want to be able to record critical traffic, e.g., routing messages, in an *always-on* fashion. What is monitored should be specified in *centralized configuration*, and we want to be able to attain a *temporally consistent* view of the recorded events in the controller domain.

Replay goals: We want to be able to replay traffic in a *coordinated* fashion across the controller domain. For replaying into a different environment or topology (e.g., in a lab environment) we want to *sub-select* traffic and potentially *map* traffic to other devices. We include *time dilation* to help investigate timer issues, create stress tests, and allow "fast forwarding" to skip over irrelevant portions of the traffic. *Bisection* of the traffic between replays can assist problem localization whereby the user repeatedly

partitions and sub-selects traffic to be replayed based on user-defined criteria (e.g., by message types), performs a test run, then continues the bisection based on whether a problem was reproducible.

(Absence of) determinism guarantees: As opposed to host-oriented replay debugging systems which strive for determinism guarantees, **OFRewind** does not – and cannot – provide strict determinism guarantees, as *black boxes* do not lend themselves to the necessary instrumentation. Instead, we leverage the insight that network device behavior is *largely deterministic* on control plane events (messages, ordering, timing). In some cases, when devices deliberately behave non-deterministically, protocol specific approaches must be taken.

6.1.3 OFRewind System Components

As seen in Figure 6.1(a), the main component of our system, **OFRewind**, runs as a proxy on the substrate control channel, i.e., between the switches and the original controller. It can thus intercept and modify substrate control plane messages to control recording and replay. It delegates recording and replay of *guest* traffic to *DataStore* components that are locally attached at regular switch ports. The number of *DataStores* attached at each switch can be chosen freely, subject to the availability of ports.

Both components can be broken down further into two modules each, as depicted in Figure 6.1(b): They consist of a recording and a replay module with a shared local storage, labeled *Ofrecord* and *Ofreplay*, and *Datarecord* and *Datareplay* respectively.

Record: *Ofrecord* captures *substrate* control plane traffic directly. When *guest* network traffic recording is desired, *Ofrecord* translates control messages to instruct the relevant switches to selectively mirror *guest* traffic to the *Datarecord* modules. **OFRewind** supports dynamic selection of the *substrate* or *guest* network traffic to record. In addition, flow-based-sampling can be used to record only a fraction of the data plane flows.

Replay: *Ofreplay* re-injects the traces captured by *Ofrecord* into the network, and thus enables domain-wide *replay debugging*. It emulates a controller towards switches or a set of switches towards a controller, and directly replays *substrate* control plane traffic. *Guest* traffic is replayed by the *Datareplay* modules, which are orchestrated by *Ofreplay*.

6.1.4 Ofrecord Traffic Selection

While it is, in principle, possible to collect full data recordings for every packet in the network, this introduces a substantial overhead both in terms of storage as well as in terms of performance. *Ofrecord*, however, allows selective traffic recording to reduce the overhead.

Selection: Flows can be classified and selected for recording. We refer to traffic *selection* whenever we make a conscious decision on what subset of traffic to record. Possible categories include: *substrate* control traffic, *guest* network control traffic, or *guest* data traffic, possibly sub-selected by arbitrary match expressions. We illustrate an example selection from these categories in Figure 6.2.

Sampling: If selection is unable to reduce the traffic sufficiently, one may apply either packet or flow sampling on either type of traffic as a reduction strategy.

Cut-offs: Another data reduction approach is to record only the first X bytes of each packet or flow. This often suffices to capture the critical meta-data and has been used in the context of intrusion detection [99].

Protocol summaries: Alternatively, it is possible to pre-process the data and extract higher level protocol information, e.g., BGP messages, HTTP protocol headers, or OpenFlow control messages.

Note that while it is possible to combine any of the above-mentioned techniques, it is not possible to rank these traffic selection strategies according to their resource consumption or replay accuracy. Sometimes cut-offs can be more accurate than protocol summaries, as they contain per packet time stamps. Other times the reverse is true, as protocol summaries may contain semantic content relevant for the scenario. When choosing a traffic selection strategy one should also keep in mind the number of devices from which to select the traffic. Moreover, it is possible to apply different selection strategies to different network components. Ultimately, the advantage of using a selection strategy that reduces traffic volume drastically is that continuous recording may be enabled and thus, help locate unexpected failures.

6.1.5 Ofreplay Operation Modes

To support testing of the different entities involved (switches, controller, end hosts) and to enable different playback scenarios, *Ofreplay* supports several different operation modes, as summarized by Table 6.1:

Name	Target	Traffic type
ctrl	Ctrl	OF msgs
switch	Switch	OF msgs
datahdr	Switch	OF msgs/ data plane traffic from headers
datafull	Switch	OF msgs/full data plane traffic

Table 6.1: *Ofreplay* operation modes

ctrl: In this operation mode, replay is directed towards the controller. *Ofreplay* plays the recorded substrate control messages from the local *storage*. This mode enables debugging of a controller application on a single developer host, without need for switches, end-hosts, or even a network. Recorded data plane traffic is not required.

switch: This operation mode replays the recorded *substrate* control messages toward the switches, reconstructing each switch's *flow table* in real time. No controller is needed, nor is *guest* traffic replayed.

datahdr: This mode uses packet headers captured by the *Datarecord* modules to regenerate the exact flows encountered at recording time, with dummy packet payloads. This enables full testing of the switch network, independent of any end hosts.

datafull: In this mode, data traffic recorded by the *DataStores* is replayed with complete payload, allowing for selective inclusion of end host traffic into the tests.

In addition to these operation modes, *Ofreplay* enables the user to further tweak the recorded traffic to match the replay scenario. Replayed messages can be *sub-selected* based on source or destination host, port, or message type. Further, message destinations can be *re-mapped* on a per-host or per-port basis. These two complementary features allow traffic to be re-targeted toward a particular host, or restricted such that only relevant messages are replayed. They enable *Ofreplay* to play recorded traffic either toward the original sources or to alternative devices, which may run a different firmware version, have a different hardware configuration, or even be of a different vendor. These features enable **OFRewind** to be used for *regression testing*. Alternately, it can be useful to map traffic of multiple devices to a single device, to perform stress tests.

In addition, the *pace* of the replay is adjustable within *Ofreplay*, enabling investigation of pace-dependent performance problems. Adjusting replay can also be used to "fast-forward" over portions of a trace, e.g., memory leaks in a switch, which typically develop over long time periods may be reproduced in an expedited manner.

6.1.6 Event Ordering and Synchronization

For some debugging scenarios, it is necessary to preserve the exact message order or mapping between *guest* and *substrate* flow data to be able to reproduce the problem. In concrete terms, the guest (data) traffic should not be replayed until the substrate (control) traffic (containing the corresponding substrate actions) has been replayed. Otherwise, *guest* messages might be incorrectly forwarded or simply dropped by the switch, as the necessary flow table state would be invalid or missing.

We do not assume tightly synchronized clocks or low latency communication channels between our **OFRewind** and the *DataStores* components. Accordingly, we cannot assume that synchronization between recorded *substrate* and *guest* flow traces, or order between flows recorded by different *DataStores* is guaranteed per se. Our design does rely on the following assumptions:

1. The *substrate* control plane channel is reliable and order-preserving.

2. The *control channel* between **OFRewind** and each individual *DataStore* is reliable and order-preserving, and has a reasonable mean latency l_c (e.g., 5 ms in a LAN setup.)

3. The *data plane* channel from **OFRewind** to the *DataStores* via the switch is not necessarily fully, but sufficiently reliable (e.g., 99.9% of messages arrive). It is not required to be order-preserving in general, but there should be some means of explicitly guaranteeing order between two messages. We define the data plane channel mean latency as l_d.

Record: Based on these assumptions, we define a logical clock C [92] on *Ofrecord*, incremented for each *substrate* control message as they arrive at *Ofrecord*. *Ofrecord* logs the value of C with each *substrate* control message. It also broadcasts the value of C to the *DataStores* in two kinds of synchronization markers: *time binning markers* and *flow creation markers*.

Time binning markers are sent out at regular time intervals i_t, e.g., every 100ms. They group flows into bins and thus constrain the search space for matching flows during replay and help reconstruct traffic characteristics within flows. Note that they do not impose a strict order on the flows within a time bin.

Flow creation markers are optionally sent out whenever a new flow is created. Based on the previous assumptions, they induce a total ordering on all flows whose creation markers have been successfully recorded. However, their usage limits the scalability of the system, as they must be recorded by all *DataStores*.

Replay: For synchronization during replay, *Ofreplay* assumes the role of a synchronization master, reading the value of C logged with the *substrate* messages. When a *DataStore* hits a synchronization marker while replaying, it synchronizes with *Ofreplay* before continuing. This assures that in the presence of *time binning markers*, the replay stays loosely synchronized between the markers (within an interval $I = i_t + l_d + l_c$). In the presence of *flow creation markers*, it guarantees that the order between the marked flows will be preserved.

6.1.7 Typical Operation

We envision that users of **OFRewind** run *Ofrecord* in an always-on fashion, always recording selected *substrate* control plane traffic (e.g., OpenFlow messages) onto a ring storage. If necessary, selected *guest* traffic can also be continuously recorded on *Datarecord*. To preserve space, low-rate control plane traffic, e.g., routing announcements, may be selected, sampling may be used, and/or the ring storage may be shrunk. When the operator (or an automated analytics tool) detects an anomaly, a replay is launched onto a separate set of hardware, or onto the production network during off-peak times. Recording settings are adapted as necessary until the anomaly can be reproduced during replay.

During replay, one typically uses some kind of debugging by elimination, either by performing binary search along the time axis or by eliminating one kind of message at a time. Hereby, it is important to choose orthogonal subsets of messages for effective problem localization.

6.1.8 Online Ofreplay

In the online replay mode *Ofrecord* and *Ofreplay* are combined. control messages are directly replayed upon arrival, e.g., to a different set of hardware or to a different *substrate* slice. Data traffic is also duplicated onto the second slice as required. This is feasible as switches already offer flexible hardware monitoring capabilities that allow one to dump the traffic of any VLAN on any port to a span (mirror) port. Therefore, the online modus allows for direct, online investigation and troubleshooting of failures in the sense of a *Mirror VNet*, as discussed in the last chapter.

When debugging live networks one has to ensure that there are sufficient network resource available for both the network under study as well as the replay target. This can be achieved by carefully selecting, e.g., a subset of the traffic for replay. In this chapter we focus on reproducing failures in a controlled environment, and leave other applications (e.g. online replay) for future study.

6.2 Implementation

In this section, we describe the implementation of **OFRewind** based on OpenFlow, selected for currently being the most widely used *split forwarding architecture*. OpenFlow is currently in rapid adoption by testbeds [70], university campuses [4], and commercial vendors [6].

OpenFlow realizes *split forwarding architecture* as an open protocol between packet-forwarding hardware and a commodity PC (the *controller*). The protocol allows the *controller* to exercise flexible and dynamic control over the forwarding behavior of OpenFlow enabled Ethernet switches at a per-flow level. The definition of a flow can be tailored to the specific application case—OpenFlow supports an 11-tuple of packet header parts, against which incoming packets can be matched, and flows classified. These range from Layer 1 (switch ports), to Layer 2 and 3 (MAC and IP addresses), to Layer 4 (TCP and UDP ports). The set of matching rules, and the actions associated with and performed on each match are held in the switch and known as the *flow table*.

We next discuss the implementation of **OFRewind**, the synchronization among the components and discuss the benefits, limitations, and best-practices of using OpenFlow to implement our system. The implementation, which is an OpenFlow controller in itself, and based on the source code of FlowVisor [136] is available under a free and open source license at [8].

6.2.1 Software Modules

To capture both the *substrate* control traffic and *guest* network traffic we use a hybrid strategy for implementing **OFRewind**. Reconsider the example shown in Figure 6.1(a) from an OpenFlow perspective. We deploy a proxy server in the OpenFlow protocol path (labeled **OFRewind**) and attach local *DataStore* nodes to the switches. The **OFRewind** node runs the *Ofrecord* and *Ofreplay* modules, and the *DataStore* nodes run *Datarecord* and *Datareplay*, respectively. We now discuss the implementation of the four software components *Ofrecord*, *Datarecord*, *Ofreplay* and *Datareplay*.

***Ofrecord*:** *Ofrecord* intercepts all messages passing between the switches and controller and applies the selection rules. It then stores the selected OpenFlow control (*substrate*) messages to locally attached data storage. Optionally, the entire flow table of the switch can be dumped on record startup. If recording of the *guest* network control and/or data traffic is performed, *Ofrecord* transforms the `FLOW-MOD` and `PACKET-OUT` commands sent from the controller to the switch to *duplicate* the packets of selected flows to a *DataStore* attached to a switch along flow path. Multiple *DataStores* can be attached to each switch, .e.g., for load-balancing. The order of

flows on the different *DataStores* in the system is retained with the help of synchronization markers. Any match rule supported by OpenFlow can be used for packet selection. Additionally, flow-based-sampling can be used to only record a fraction of the flows.

***Datarecord*:** The *Datarecord* components located on the *DataStores* record the selected guest traffic, as well as synchronization and control metadata. They are spawned and controlled by *Ofrecord*. Their implementation is based on `tcpdump`, modified to be controlled by *Ofrecord* via a TCP socket connection. Data reduction strategies that cannot be implemented with OpenFlow rules (e.g., packet sampling, cut-offs) are executed by *Datarecord* before writing the data to disk.

***Ofreplay*:** *Ofreplay* re-injects OpenFlow control plane messages as recorded by *Ofrecord* into the network and orchestrates the guest traffic replay by the *Datareplay* components on the *DataStores*. It supports replay towards the *controller* and *switches*, and different levels of data plane involvement (*switch*, *datahdr*, *datafull*, see Section 6.1.5.) Optionally, a flow table dump created by *Ofrecord* can be installed into the switches prior to replay. It supports traffic *sub-selection* and *mapping* towards different hardware and *time dilation*.

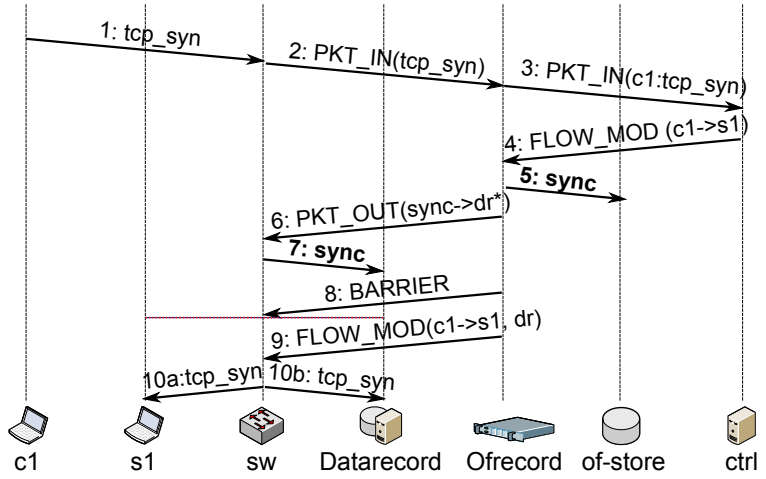
***Datareplay*:** The *Datareplay* components are responsible for re-injecting guest traffic into the network. They interact with and are controlled by *Ofreplay* for timing and synchronization. The implementation is based on `tcpreplay`. Depending on the record and replay mode, they reconstruct or synthesize missing data before replay, e.g., dummy packet payloads, when only packet headers have been recorded.

6.2.2 Synchronization

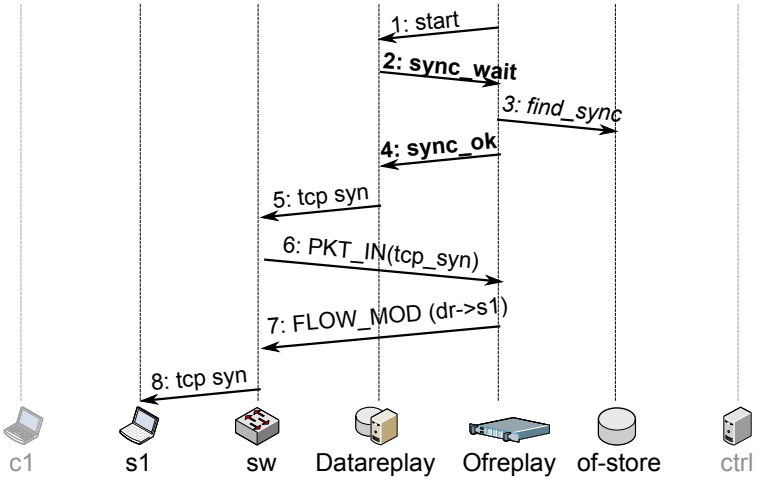
As we do not assume precise time synchronization between *Ofrecord* and the *DataStores*, the implementation uses *time binning markers* and *flow creation markers*, as discussed in Section 6.1.6. These are packets with unique ids flooded to all *DataStores* and logged by *Ofrecord*. The ordering of these markers relative to the recorded traffic is ensured by OpenFlow `BARRIER` messages[1]. We now discuss by example how the markers are used.

Record synchronization: Figure 6.3(a) illustrates the use of *flow creation markers* during recording. Consider a simple *Ofrecord* setup with two hosts *c1* and *s1* connected to switch *sw*. The switch is controlled by an instance of *Ofrecord*, which in turn acts as a client to the network controller *ctrl*. *Ofrecord* records to the local storage *of-store*, and controls an instance of *Datarecord* running on a *DataStore*. Assume that a new TCP connection is initiated at *c1* toward *s1*. The following chain of events ensures the correct synchronization of new flows during recording:

[1] A BARRIER message ensures that all prior OpenFlow messages are processed before subsequent messages are handled. In its absence, messages may be reordered.



(a) *Ofrecord*



(b) *Ofreplay*

Figure 6.3: *DataStore* synchronization mechanism in **OFRewind**

Step 1: *c1* generates a *tcp syn* packet, sent to the switch.

Step 2: As no matching flow table entry exists, *sw* sends *msg1*, an OpenFlow `PACKET-IN` to *Ofrecord*.

Step 3: *Ofrecord* in turn relays it to *ctrl*.

Step 4: *Ctrl* may respond with *msg2*, a `FLOW-MOD` message.

Step 5: To enable synchronized replay and reassembly of the control and data records, *Ofrecord* now creates a flow creation marker (*sync*), containing a unique id, the current time, and the matching rule of *msg1* and *msg2*. Both *msg1* and *msg2* are then annotated with the id of *sync* and saved to *of-store*.

Step 6: *Ofrecord* then sends out 3 messages to *sw1*: first, a `PACKET-OUT` message containing the *flow creation marker* sent to *all DataStores*.

Step 7: This `PACKET-OUT` message prompts the switch to send out *sync* to all its attached *DataStores*.

Step 8: The second message sent from *Ofrecord* is a `BARRIER` message, which ensures that the message from step 7 is handled before any subsequent messages.

Step 9: *Ofrecord* sends a modified `FLOW-MOD` message directing the flow to both the original receiver, as well as *one DataStore* attached to the switch.

Step 10: This prompts the switch to output the flow both to *s1* (step 10a) and *DataStore* (step 10b).

Replay synchronization: For synchronizing replay, *Ofreplay* matches data plane events to control plane events with the help of the flow creation markers recorded by *Ofrecord*. Consider the example in Figure 6.3(b). Based on the previous example, we replay the recording in *data plane mode* towards the switch *sw* and host *s1*.

Step 1: *Ofreplay* starts *Datareplay* playback on the *DataStore*.

Step 2: *Datareplay* hits the flow creation marker *sync*, then sends a `sync_wait` message to the controller, and goes to sleep.

Step 3: *Ofrecord* continues replay operation, until it hits the corresponding flow creation marker *sync* on the *of-store*.

Step 4: Then, it signals *Datareplay* to continue with a `sync_ok` message.

Step 5: *Datareplay* outputs the packet to the switch.

Step 6: This generates a `PACKET-IN` event.

Step 7: *Ofreplay* responds with the matching `FLOW-MOD` event from the log.

Step 8: This installs a matching flow rule in the switch and causes the flow to be forwarded as recorded.

6.2.3 Discussion

We now discuss the limitations imposed by OpenFlow on our implementation, and best practices for avoiding replay inaccuracies.

OpenFlow limitations: While OpenFlow provides a useful basis for implementing **OFRewind**, it does not support all the features required to realize all operation modes. OpenFlow does not currently support **sampling** of either flows or packets. Thus, *flow sampling* is performed by *Ofrecord*, and packet sampling is performed by *Datarecord*. This imposes additional load on the channel between the switches and the *DataStores* for data that is not subsequently recorded. Similarly, the OpenFlow data plane does not support forwarding of **partial packets**[2]. Consequently, full packets are forwarded to the *DataStore* and only their headers may be recorded. OpenFlow also does not support automatic **flow cut-offs** after a specified amount of *traffic*[3]. The cut-off can be performed in the *DataStore*. Further optimizations are possible, e.g., regularly removing flows that have surpassed the threshold.

Avoiding replay inaccuracies: To reliably reproduce failures during replay in a controlled environment, one must ensure that the environment is properly initialized. We suggest therefore, to use the flow table dump feature and, preferably, reset (whenever possible) the switches and controller state before starting the replay operation. This reduces any unforeseen interference from previously installed bad state.

When bisecting during replay, one must consider the interdependencies among message types. `FLOW-MOD` messages are for example, responsible for creating the flow table entries and their arbitrary bisection may lead to incomplete or nonsense forwarding state on the switch.

Generally speaking, replay inaccuracies can occur when: (a) the chain of causally correlated messages is recorded incompletely, (b) synchronization between causally correlated messages is insufficient, (c) timers influence system behavior, and (d) network communication is partially non-deterministic. For (a) and (b), it is necessary to adapt the recording settings to include more or better-synchronized data. For (c) a possible approach is to *reduce* the traffic being replayed via sub-selection to reduce stress on the devices and increase accuracy. We have not witnessed this problem in our practical case studies. Case (d) requires the replayed traffic to be modified. If the non-determinism stems from the transport layer (e.g., TCP random initial sequence numbers), a custom transport-layer handler in *Datareplay* can shift sequence numbers accordingly for replay. For application non-determinism (e.g., cryptographic nonces), application-specific handlers must be used.

[2]It *does* support a cut-off for packets forwarded to the *controller*.
[3]Expiration after a specified amount of *time* is supported.

Case study	Class	OF-specific
Switch CPU Inflation	black box (switch)	no
Anomalous Forwarding	black box (switch)	yes
Invalid Port Translation	OF controller	yes
NOX Parsing Error	OF controller	yes
Faulty Route Advertisements	software router	no

Table 6.2: Overview of the case studies

When the failure observed in the production network does not appear during replay, we call this a **false negative** problem. When the precautions outlined above have been taken, a repeated *false negative* indicates that the failure is likely not triggered by network traffic, but other events. In a **false positive** case, a failure is observed during replay which does not stem from the same root cause. Such inaccuracies can often be avoided by careful comparison of the symptoms and automated repetition of the replay.

6.3 Case Studies

In this section, we demonstrate the use of **OFRewind** for localizing problems in *black box network devices*, *controllers*, and *other software components*, as summarized in Table 6.2. We examine two switch problems, *switch CPU inflation* (6.3.2) and *anomalous forwarding* (6.3.4), two controller problems, *invalid port translation* (6.3.5) and *NOX PACKET-IN parsing error* (6.3.6), and a non-OpenFlow related software problem. concerning *faulty route advertisements* (6.3.7). These case studies also demonstrate the benefits of *bisecting* the control plane traffic (6.3.2), of *mapping* replays onto different pieces of hardware (6.3.4), from a production network onto a developer machine (6.3.6), and the benefit of a *temporally consistent* recording of multiple switches (6.3.7).

6.3.1 Experimental Setup

For our case studies we use a network with switches from three vendors: **Vendor A**, **Vendor B**, **Vendor C**. Each switch has two PCs connected to it. Figure 6.4 illustrates the connectivity. All switches in the network have a control-channel to *Ofrecord*. *DataStores* running *Datarecord* and *Datareplay* are attached to the switches as necessary. We use NOX [110], unless specified otherwise, as the high level controller performing the actual routing decisions. It includes the *routing* module, which performs shortest path routing by learning the destination MAC address, and the $spanning - tree$ module, which prevents broadcast storms in the network by using Link Layer Discovery Protocol (LLDP) to identify if there is a loop in the network. All OpenFlow applications, viz. NOX, FlowVisor, *Ofreplay*, and *Ofrecord*, are run on the same server.

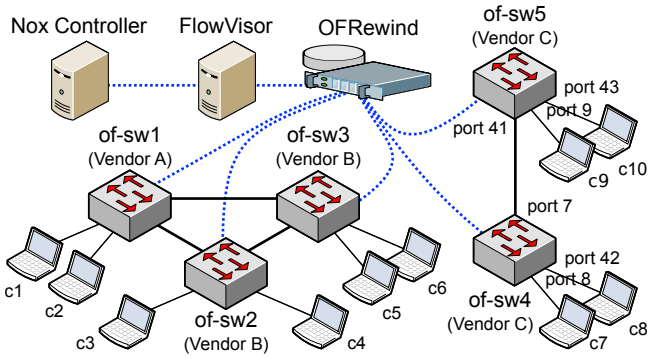
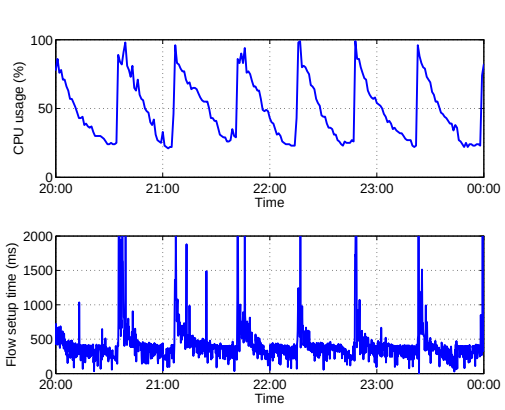


Figure 6.4: Overview of the lab environment for case studies

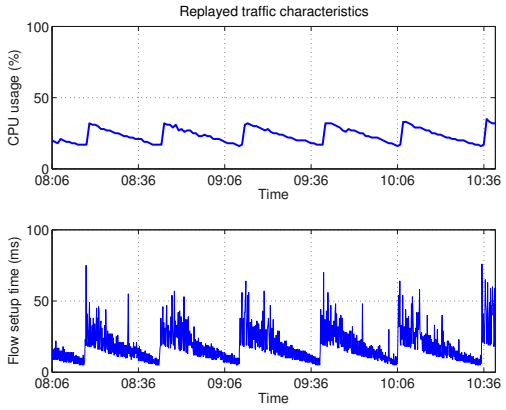
6.3.2 Switch CPU Inflation

Figure 6.5 shows our attempt at reproducing the CPU oscillation we reported in Section 6. As stated earlier, there is no apparent correlation between the ingress traffic and the CPU utilization. We record all control traffic in the production network, as well as the traffic entering/exiting the switch, while the CPU oscillation is happening. Figure 6.5(a) shows the original switch performance during recording. We, then, iteratively replay the corresponding control traffic over a similar switch in our isolated experimental setup. We replay the recorded data traffic to 1 port of the switch and connect hosts that send `ICMP` datagrams to the other ports. In each iteration, we have *Ofreplay bisect* the trace by OpenFlow message type, and check whether the symptom persists. When replaying the port and table statistic requests, we observe the behavior as shown in Figure 6.5(b). Since the data traffic is synthetically generated, the amplitude of the CPU oscillation and the flow setup time variation is different from that in the original system. Still, the sawtooth pattern is clearly visible. This successful reproduction of the symptom helps us identify the issue to be related to port and table statistics requests. Note that these messages have been causing the anomaly, even though their arrival rate (24 messages per minute) is not in any way temporally correlated with the perceived symptoms (30-minute CPU sawtooth pattern). We reported this issue to the vendor, since at this point we have no more visibility into the switch software implementation.

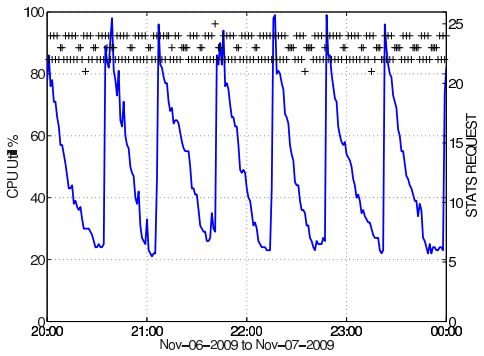
OFRewind thus, has proved useful in localizing the cause for the anomalous behavior of a black box component that would otherwise have been difficult to debug. Even though the bug in this case is related to a prototype OpenFlow device, the scenario as such (misbehaving black box component) and approach (debugging by replay and bisection of control-channel traffic) are arguably applicable to non-OpenFlow cases as well.



(a) Switch performance with original traffic



(b) Switch performance with replayed traffic



(c) `STATS-REQUEST` vs. CPU

Figure 6.5: Sawtooth CPU pattern case study: pattern reproduced during replay of port and table `STATS-REQUEST` messages. Figure (c) shows no observable temporal correlation to message arrivals.

6.3.3 Broadcast Storms

Many Ethernet networks use redundant links for fault tolerance. As loops in the active Ethernet topology can cause broadcast storms that make the network dysfunctional, the *Spanning tree protocol* (STP) is used to discover a loop free subgraph to be used. STP periodically sends specific datagrams—Bridge Protocol Data Units (BPDU) for legacy networks or Link Layer Discovery Protocol (LLDP) packets for OpenFlow-enabled networks—to learn about existing links and thus about possible loops. STP heavily depends on the correct forwarding of these special datagrams. Incorrect handling can lead to STP convergence problems. In the Stanford OpenFlow deployment, two such variations have been encountered: (a) legacy switches dropping the LLDPs and (b) OpenFlow switches dropping the BPDUs.

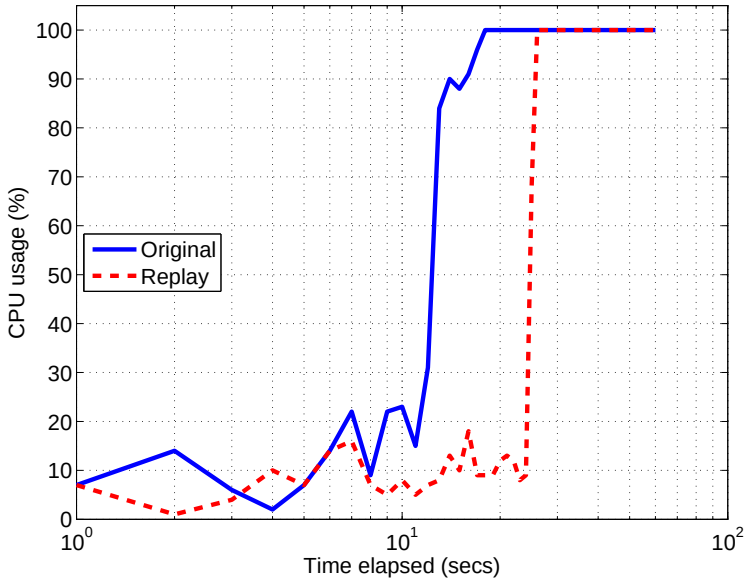


Figure 6.6: Broadcast storm case study: Time series of CPU utilization, during the replay of a broadcast storm in networks where STP does not converge. The CPU spikes are out of sync because of non-determinism in the switch backoff period while connecting to the controller.

We show how we are able to reproduce case (a) in the experimental network. When the LLDPs are dropped by a legacy switch, the controller becomes oblivious to the loop. Thus a subsequently arriving broadcast ARP packet causes a broadcast storm, pushing the CPU usage of the switch (which does not enforce strict CPU isolation) to hit 100%. Figure 6.6 shows the CPU utilization while the failure occurred as well as during replay [4].

We start our troubleshooting by turning off OpenFlow on `of-sw1`, effectively transforming it into a legacy switch, and thus creating a loop in the network (triangle between `of-sw1`, `of-sw2` and `of-sw3`). Then, we only replay the non-`FLOW-MOD` control-messages. The loop, in this run, is not triggered as any broadcast traffic is buffered and then dropped. However, once we include the `FLOW-MOD` messages, the `FLOOD` action rule causes the broadcast ARP traffic to be forwarded forever. As this forwarding is currently done by the CPU this causes the CPU utilization spike and imposes a denial of service attack for all other flows.

By making this **OFRewind** test case part of a regression test suite, we can now avoid broadcast storms in production networks. Further, this test case can also be used to verify any new switch software release and in particular to test the convergence of STP.

[4]The original run and the replayed one occurred at two different time instances separated by 10 minutes.

Counts	Match
duration=181s	in_port=8
n_packets=0	dl_type=arp
n_bytes=3968	dl_src=00:15:17:d1:fa:92
idle_timeout=60	dl_dst=ff:ff:ff:ff:ff:ff
hard_timeout=0	actions=`FLOOD`

Table 6.3: Anomalous forwarding case study: switch flow table entry, during replay.

6.3.4 Anomalous Forwarding

To investigate the performance of devices from a new vendor, **Vendor C**, we record the substrate and guest traffic for a set of flows, sending 10 second delayed `ping` between a pair of hosts attached to the switch from **Vendor B** (`of-sw3` in Figure 6.4). We then use the device/port mapping feature of *Ofreplay* to play back traffic from $c7$ to $c8$ over port 8 and port 42 belonging to the switch from **Vendor C**, in Figure 6.4.

Upon replay, we observe an interesting limitation of the switch from **Vendor C**. The `ping` flow stalls at the ARP resolution phase. The ARP packets transmitted from host $c7$ are received by host $c10$, but not by $c8$ nor $c9$. The flow table entry created in `of-sw4` during replay, is shown in Table 6.3, similar to that created during the original run. We conclude that the `FLOOD` action is not being properly applied by the switch from **Vendor C**.

Careful observation reveals that traffic received on a "low port" (one of the first 24 ports) to be flooded to any "high ports" (last 24 ports) and vice-versa is not flooded correctly. Effectively, the flood is restricted within a 24 port group within the switch (lower or higher). This fact has been affirmed by the vendor, confirming the successful debugging.

We additionally perform the replay after adding static ARP entries to the host $c7$. In this case, we observe that flow setup time for the subsequent unicast `ping` traffic on **Vendor C** is consistently higher than that observed for **Vendor B** and **Vendor A** switches. This indicates that **OFRewind** has further potential in profiling switches and controllers.

6.3.5 Invalid Port Translation

In this case study, we operate *Ofreplay* in the *ctrl* mode in order to debug a controller issue. The controller we focus on is the publicly available FlowVisor [136].

FlowVisor (FV) is a special purpose OpenFlow controller that acts as a proxy between multiple OpenFlow switches and controllers (*guests*), and thus assumes the role of a hypervisor for the OpenFlow control plane (see Figure 6.4). To this end, the overall

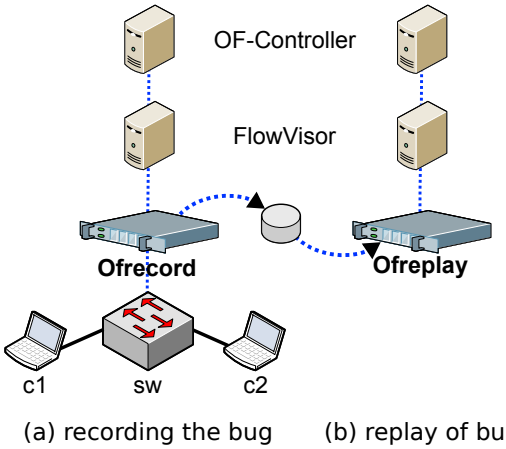


Figure 6.7: Invalid port translation case study: Schematic overview

flow space is partitioned by FV into distinct classes, e.g., based on IP addresses and ports, and each guest is given control of a subset. Messages between switches and controllers are then filtered and translated accordingly.

We investigate an issue in where the switch from **Vendor C** works fine with the NOX controller, but not through the FV. We record the OpenFlow control plane traffic from the switch to FV in our production setup, as seen on the left side of Figure 6.7. We then replay the trace on a developer system, running *Ofreplay*, FV and the upstream controller on a single host for debugging. *Ofreplay* thus assumes the role of the switch.

Through repeated automated replay of the trace on the development host, we track down the source of the problem: It is triggered by a switch announcing a non-contiguous range of port numbers (e.g., 1, 3, 5). When FV translates a `FLOOD` action sent from the upstream controller to such a switch, it incorrectly expands the port range to a contiguous range, including ports that are not announced by the switch (e.g., 1, **2**, 3, **4**, 5). The switch then drops the invalid action.

Here, **OFRewind** proves useful in localizing the root cause for the failure. Replaying the problem in the development environment enables much faster turnaround times, and thus reduces debugging time. Moreover, it can be used to verify the software patch that fixes the defect.

6.3.6 NOX `PACKET-IN` Parsing Error

We now investigate a problem, reported on the NOX [110] development mailing list, where the NOX controller consistently drops the ARP reply packet from a specific host. The controller is running the `pyswitch` module.

The bug reporter provides a `tcpdump` of the traffic between their switch and the controller. We verify the existence of the bug by replaying the control traffic to our instance of the NOX. We then gradually increase the debug output from NOX as we play back the recorded OpenFlow messages to NOX.

Repeating this processes reveals the root cause of the problem: NOX deems the destination MAC address `00:26:55:da:3a:40` to be invalid. This is because the MAC address contains the byte `0x3a`, which happens to be the binary equivalent of the character ':' in ASCII. This "fake" ASCII character causes the MAC address parser to interpret the MAC address as ASCII, leading to a parsing error and the dropped packet. Here, *Ofreplay* provides the necessary debugging context to faithfully reproduce a bug encountered in a different deployment, and leads us to the erroneous line of code.

6.3.7 Faulty Routing Advertisements

In a departure from OpenFlow network troubleshooting, we examine how **OFRewind** can be used to troubleshoot more general, event-driven network problems. We consider the common problem of a network suffering from a mis-configured or faulty router. In this case, we demonstrate how **OFRewind** can be advantageously used to identify the faulty component.

We apply **OFRewind** to troubleshoot a documented bug (Quagga Bugzilla #235) detected in a version of the RIPv1 implementation of the *Quagga* [119] software routing daemon. In the network topology given by Figure 6.8, a network operator notices that shortly after upgrading *Quagga* on software router B, router C subsequently loses connectivity to Network 1. As routing control plane messages are a good example of low-volume guest control plane traffic, they can be recorded by *Ofrecord* always-on or, alternatively, as a precautionary measure during upgrades. Enabling *flow creation sync markers* for the low-volume routing control plane messages ensures the global ordering is preserved.

The observation that router C loses its route to Network 1 while router B maintains its route, keys the operator to record traffic arriving at and departing from B. An analysis of the *Ofrecord* flow summaries reveals that although RIPv1 advertisements arrive at B from A, no advertisements leave B toward C. Host-based debugging of the RIPd process can then be used on router B in conjunction with *Ofreplay* to replay the trigger sequence and inspect the RIPd execution state. This reveals the root cause of the bug – routes toward Network 1 are not announced by router B due to this (0.99.9) version's handling of classful vs. CIDR IP network advertisements – an issue inherent to RIPv1 on classless networks.

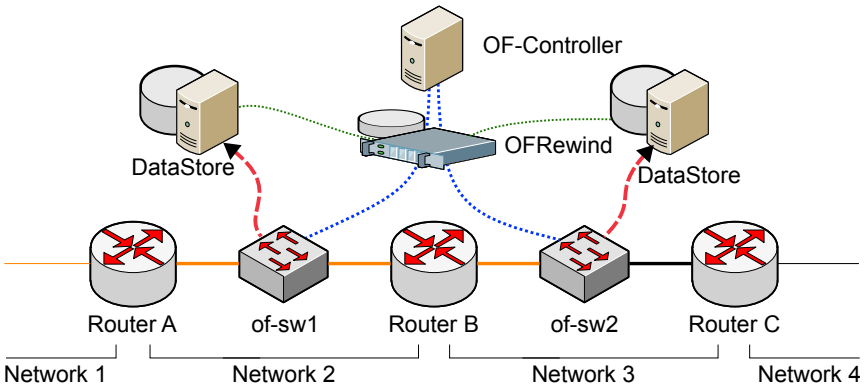


Figure 6.8: Faulty route advertisements case study: Setup

6.3.8 Discussion

Without making any claims regarding the representativeness of the workload or switch behavior, in this limited space, we highlight in these case studies, the principle power and flexibility of **OFRewind**. We observe that **OFRewind** is capable of replaying subpopulations of control or data traffic, over a select network topology (switches and ports) or to select controllers, in a sandbox or production environment.

We further note that **OFRewind** has potential in switch (or controller) benchmarking. By creating a sandbox for experimentation that can be exported to a standard replay format, a network operator can concretely specify the desired behavior to switch (or controller) design engineers. The sandbox can then be run within the premises of the switch (or controller software) vendor on a completely new set of devices and ports. On receiving the device (or software), the network operator can conduct further benchmarking to compare performance of different solutions in a fair manner.

Comparison with traditional recording: Based on the case presented in the last section, we compare the effort of recording and instrumenting the network with and without **OFRewind**. Note that while the specific traffic responsible for the failure is small (RIP control plane messages), the total traffic volume on the involved links may be substantial. To attain a synchronized recording of this setup without **OFRewind**, and in the absence of host-based instrumentation, one has to (1) deploy monitoring servers that can handle the *entire* traffic on each link of interest, e.g., [105], and redeploy as link interests change. Then, one must either (2a) reconfigure both switches to enable span ports (often limited to 2 on mid-range hardware) or (2b) introduce a tap into the physical wiring of the networks. Manually changing switch configurations runs a risk of operator error and introducing a tap induces downtime and is considered even riskier. (3) Additionally, the monitoring nodes may

of-simple	reference controller emulating a learning switch
nox-pyswitch	NOX controller running Python `pyswitch` module
nox-switch	NOX controller running C-language `switch` module
flowvisor	Flowvisor controller, running a simple allow-all policy for a single guest controller
ofrecord	*Ofrecord* with substrate mode recording
ofrecord-data	*Ofrecord* with guest mode recording, with one data port and sync beacons and barriers enabled

Table 6.4: Evaluation: Overview of controllers used

have to be synced to microsecond level to keep the flows globally ordered, requiring dedicated, expensive hardware. With **OFRewind**, one requires only a few commodity PCs acting as *DataStores*, and a single, central configuration change to record a *consistent* traffic trace. controller software) vendor on a completely new set of devices and ports. On receiving the device (or software), the network operator can conduct further benchmarking to compare performance of different solutions in a fair manner.

6.4 Evaluation

When deploying **OFRewind** in a live production environment, we need to pay attention to its scalability, overhead and load on the switches. This section quantifies the general impact of deploying *Ofrecord* in a production network, and analyzes the replay accuracy of *Ofreplay* at higher flow rates.

6.4.1 Ofrecord Controller Performance

A key requirement for practical deployment of *Ofrecord* in a production environment is recording performance. It must record fast enough to prevent a performance bottleneck in the controller chain.

Using `cbench` [42], we compare the controller performance exhibited by several well known controllers, listed in Table 6.4. *Of-simple* and NOX are stand-alone controllers, while *flowvisor* and *ofrecord* act as a proxy to other controllers. *Ofrecord* is run twice: in substrate mode (*ofrecord*), recording the OpenFlow substrate traffic, and in guest mode (*ofrecord-data*), additionally performing OpenFlow message translations and outputting sync marker messages. Note that no actual guest traffic is involved in this experiment.

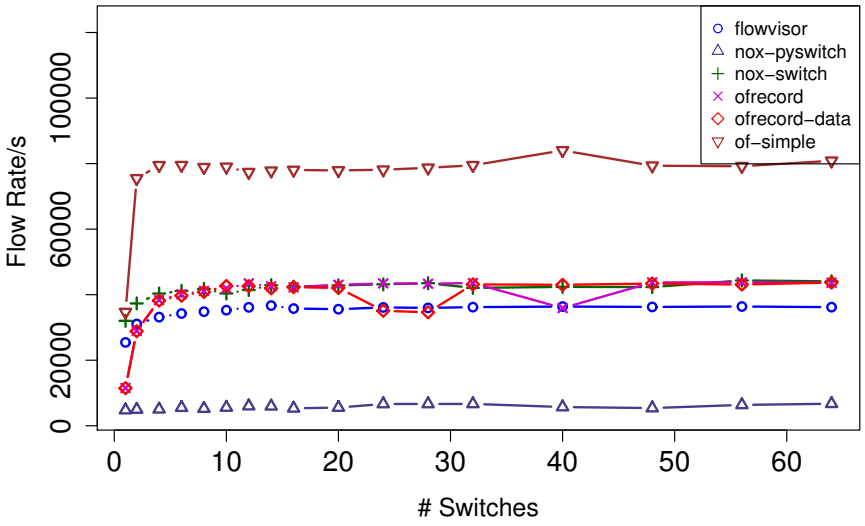


Figure 6.9: Controller performance: # switches vs. median flow rate throughputs for different controllers using cbench.

The experiment is performed on a single commodity server (Intel Xeon L5420, 2.5 GHz, 8 cores, 16 GB RAM, 4xSeagate SAS HDDs in a RAID 0 setup). We simulate, using `Cbench`, 1-64 switches connecting to the controller under test, and send back-to-back `PACKET-IN` messages to measure the maximum flow rate the controller can handle. `Cbench` reports the current flow rate once per second. We let each experiment run for 50 seconds, and remove the first and last 4 results for warmup and cool-down.

Figure 6.9 presents the results. We first compare the flow rates of the stand-alone controllers. *Nox-pyswitch* exhibits a median flow rate of *5,629* flows/s over all switches, *nox-switch* reports *42,233* flows/s, and *of-simple* *78,908* flows/s. Consequently, we choose *of-simple* as the client controller for the proxy controllers. We next compare *flowvisor* and *ofrecord*. *Flowvisor* exhibits a median flow throughput of *35,595* flows/s. *Ofrecord* reports *42,380* flows/s, and *ofrecord-data* reports *41,743*. There is a slight variation in the performance of *ofrecord*, introduced by the I/O overhead. The minimum observed flow rates are *28,737* and *22,248*. We note that all controllers exhibit worse maximum throughput when only connected to a single switch, but show similar performance for $2 - 64$ switches.

We conclude that *Ofrecord*, while outperformed by *of-simple* in control plane performance, is unlikely to create a bottleneck in a typical OpenFlow network, which often includes a FlowVisor instance and guest domain controllers running more complex policies on top of NOX. Note that all controllers except *nox-pyswitch* perform an order of magnitude better than the maximum flow rates supported by current prototype OpenFlow hardware implementations (max. *1,000* flows/s).

6.4.2 Switch Performance During Record

When *Ofrecord* runs in *guest mode*, switches must handle an increase in OpenFlow control plane messages due to the sync markers. Additionally, `FLOW-MOD` and `PACKET-OUT` messages contain additional actions for mirroring data to the *DataStores*. This may influence the *flow arrival rate* that can be handled by a switch.

To investigate the impact of *Ofrecord* deployment on switch behavior, we use a test setup with two 8-core servers with 8 interfaces each, wired to two prototype OpenFlow hardware switches from **Vendor A** and **Vendor B**. We measure the supported flow arrival rates by generating minimum sized UDP packets with increasing destination port numbers in regular time intervals. Each packet thus creates a new flow entry. We record and count the packets at the sending and the receiving interfaces. Each run lasts for 60 seconds, then the UDP packet generation rate is increased.

Figure 6.10 presents the flow rates supported by the switches when controlled by *ofrecord*, *ofrecord-data*, and *of-simple*. We observe that all combinations of controllers and switches handle flow arrival rates of at least 200 flows/s. For higher flow rates, the **Vendor B** switch is CPU limited and the additional messages created by *Ofrecord* result in reduced flow rates (*ofrecord*: 247 flows/s, *ofrecord-data*: 187) when compared to *of-simple* (393 flows/s). **Vendor A** switch does not drop flows up to an ingress rate of 400 flows/s. However, it peaks at 872 flows/s for *ofrecord-data*, 972 flows/s for *ofrecord* and 996 flows/s for *of-simple*. This indicates that introducing *Ofrecord* imposes an acceptable performance penalty on the switches.

6.4.3 DataStore Scalability

Next we analyze the scalability of the *DataStores*. Note that *Ofrecord* is not limited to using a single *DataStore*. Indeed, the aggregate data plane traffic (T_s bytes in c_F flows) can be distributed onto as many *DataStores* as necessary, subject to the number of available switch ports. We denote the number of *DataStores* with n and enumerate each *DataStore* D_i subject to $0 \leq i < n$. The traffic volume assigned to each *DataStore* is T_i, such that $T_s = \sum T_i$. The flow count on each *DataStore* is c_i.

The main overhead when using *Ofrecord* is caused by the *sync markers* that are flooded to *all DataStores* at the same time. Thus, their number limits the scalability of the system. *Flow-sync markers* are minimum-sized Ethernet frames that add constant overhead ($\theta = 64B$) per new flow. Accordingly, the absolute overhead for each *DataStore* is: $\Theta_i = \theta \cdot c_i$. The absolute overhead for the entire system is $\Theta = \sum \Theta_i = \theta \cdot c_F$, the relative overhead is: $\Theta_{rel} = \frac{\Theta}{T_s}$.

In the Stanford production network, of four switches with one *DataStore* each, a 9 hour day period on a workday in July 2010 generated $c_F = 3,891,899$ OpenFlow messages that required synchronization. During that period, we observed 87.977 GB

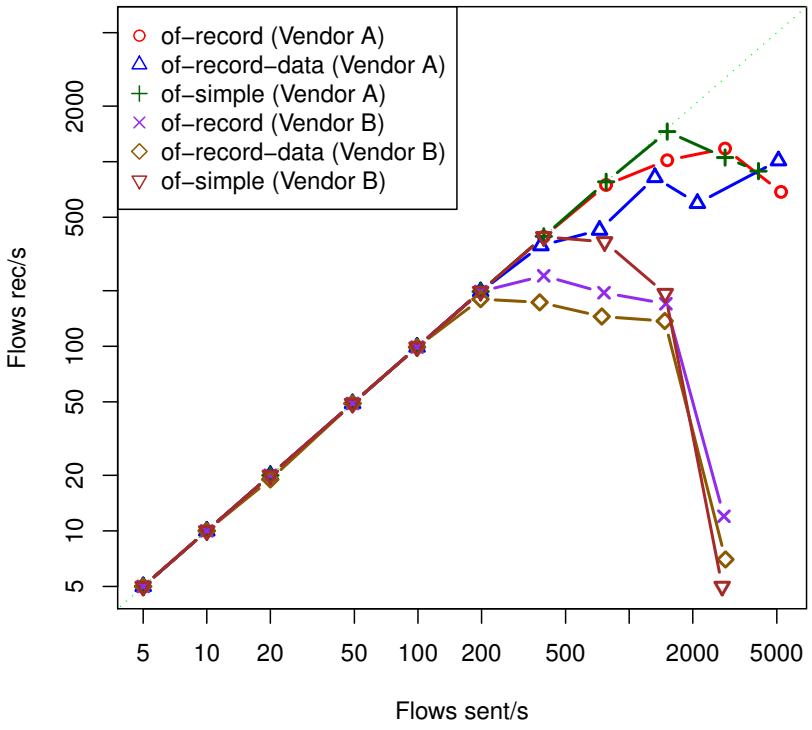


Figure 6.10: Switch performance: Mean rate of flows sent vs. successfully received with controllers *ofrecord*, *ofrecord-data*, and *of-simple* and switches from **Vendor A** and **B**.

of data plane traffic. Thus, the overall relative overhead is $\Theta_{rel} = 1.13\%$, small enough to not impact the capacity, and allow scaling up the deployment to a larger number of *DataStores*.

6.4.4 End-to-End Reliability And Timing

We now investigate the end-to-end reliability and timing precision of **OFRewind** by combining *Ofrecord* and *Ofreplay*. We use minimum size flows consisting of single UDP packets sent out at a uniform rate to obtain a worst-case bound. We vary the flow rate to investigate scalability. For each run, we first record the traffic with *Ofrecord* in guest mode with flow sync markers enabled. Then, we play back the trace and analyze the end-to-end drop rate and timing accuracy. We use a two-server setup connected by a single switch of **Vendor B**. Table 6.5 summarizes the results. Flow rates up to 200 Flows/s are handled without drops. Due to the flow sync markers, no packet reorderings occur and all flows are replayed in the correct order. The exact inter-flow timings vary though, upwards from 50 Flows/s.

To investigate the timing accuracy further, we analyze the relative deviation from the expected inter-flow delay. Figure 6.11 presents the deviations experienced by the flows during the different phases of the experiment. Note that while there are certainly

Rate (Flows/s)	Drop %	sd(timing, in ms)
5	0 %	4.5
10	0 %	15.6
20	0 %	21.1
50	0 %	23.4
100	0 %	10.9
200	0 %	13.9
400	19%	15.8
800	41 %	21.5

Table 6.5: End-to-end performance measurement with uniformly spaced flows consisting of 1 UDP packet

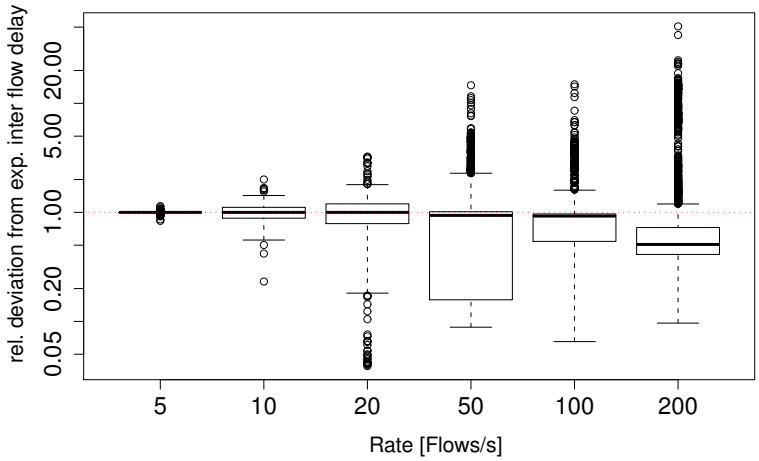


Figure 6.11: End-to-end flow timing accuracy: boxplot of the relative deviation from expected inter-flow delay.

outliers for which the timing is far off, the *median* inter-flow delay remains close to the optimum for up to 100 Flows/s. Higher rates show room for improvement.

6.4.5 Scaling Further

We now discuss from a theoretical standpoint the limits of scalability intrinsic to the design of **OFRewind** when scaling beyond currently available production networks or testbeds. As with other OpenFlow-based systems, the performance of **OFRewind** is limited by the switch flow table size and the switch performance when updating and querying the flow table. We observe these to be the most common performance bottlenecks in OpenFlow setups today. Controller domain scalability is limited by the capacity of the link that carries the OpenFlow control channel, and the network and CPU performance of the controller. Specific to **OFRewind**, the control plane components require sufficient I/O performance to record the selected OpenFlow mes-

sages – not a problem in typical developments. When recording data plane network traffic, *DataStore* network and storage I/O capacity must be sufficient to handle the aggregate throughput of the selected flows. As load-balancing is performed over *DataStores* at flow granularity, **OFRewind** cannot fully record individual flows that surpass the network or storage I/O capacity of a single *DataStore*. When *flow creation markers* are used for synchronization, the overhead grows linearly with the number of *DataStores* and the number of flows. Thus, when the average flow size in a network is small, and synchronization is required, this may limit the scalability of a controller domain. For scaling further, **OFRewind** may in the future be extended to a *distributed controller domain*. While a quantitative evaluation is left for future study, we note that the lock-step approach taken to coordinate the replay of multiple instances of *Datarecord* and *Datareplay* (see Section 6.1.6) can be extended to synchronize multiple instances of **OFRewind** running as proxies to instances of a distributed controller. The same trade-offs between accuracy and performance apply here as well.

6.5 Related Work

Our work builds on a wealth of related work in the areas of recording and summarizing network traffic, replay debugging based on networks and on host primitives, automated problem diagnosis, pervasive tracing, and testing large-scale systems.

Recording/summarizing network traffic: Apart from the classical tool *tcpdump*[10], different approaches have been suggested in the literature to record high-volume network traffic. Anderson et al. [27] records at kernel-level to provide bulk capture at high rates, and Antonelli et al. [29] focuses on long-term archival and stores traffic on *tapes*. Hyperion [52] employs a dedicated stream file system to store high-volume data streams, indexing stream data using Bloom filters. While these systems are not aimed at reproducing network failures their basic methodology may be used to gather data for **OFRewind**. Another approach is to store higher-level abstractions of the network traffic to reduce the data volume. Reiss et al. [123] record flows and provide real-time query facilities; Shanmugasundaram et al. [133] record key events of activity such as connections and scans; and [44, 103] both provide frameworks suitable for performing different data reduction techniques. Cooke et al. [49] aggregate data as it ages: first packets are stored; these are than transformed into flows. These systems are aimed at gathering network statistics rather than reproducing network failures. Reducing traffic volume by omitting parts of the traffic is employed by the Shunt [74] and the time-machine [99]. Their focus is to support intrusion detection and intrusion prevention. Our selection strategies borrow many of their ideas, more can be incorporated for improved performance. Note that these systems do not target network replay, and that all integrated control and data plane monitoring systems

face scalability challenges when monitoring high-throughput links as the monitor has to consider all data plane traffic, even if only a subset is to be recorded.

Similar to our approach of recording in a split-architecture environment, *OpenSafe* [33] leverages OpenFlow for flexible network monitoring but does not target replay or provide temporal consistency among multiple monitors. Complementary to our work, *OpenTM* [148] uses OpenFlow statistics to estimate the traffic matrix in a controller domain. *MeasuRouting* [121] enables flexible and optimized placement of traffic monitors with the help of OpenFlow, and could facilitate non-local *DataStores* in **OFRewind**.

Network replay debugging: None of the mentioned systems provide replay capabilities for networks. In this sense, the closest siblings to our work are the classical tools *tcpdump* and *tcpreplay* [11]. In fact, **OFRewind** uses these tools internally for data plane recording and replay, but significantly adds to their scope, scalability, and coherence: It records from a controller domain instead of a single network interface, can *select* traffic on the control plane and *load-balance* multiple *DataStores* for scalability, and can record a *temporally consistent* trace of the controller domain.

Replay debugging based on host primitives: Complementary to our network based replay, there exists a wealth of approaches that enable replay debugging for end-hosts. Some systems use instrumentation based on host virtualization technologies to provide a controlled environment for recording and then replaying non-deterministic events [56, 89]. Others rely on instrumentation via preloaded libraries [69], or extend the concept to distributed applications and aim to provide a global debugger [68]. A common limitation of these systems is that they are unable to trace parallel threads in today's multi-core systems, which limits their practical use. Different approaches have been proposed to overcome this limitation: Altekar et.al. [20] propose to relax consistency requirements during recording and infer causality later. According to currently reported results, inference times can take up to hundred times the duration of the actual runs. Others rely on hardware assistance for high-performance parallelism support [82, 106], which incurs little overhead, but limits deployment. DCR [21], a recent approach, emphasizes the importance of the control plane for debugging. These systems provide fully deterministic replay capabilities important for debugging complex end-host systems. Parallelism remains a challenging problem and they typically cannot be used for *black box* network components.

Automated problem diagnosis: A deployment of **OFRewind** can be complemented by a system that focuses on automated problem diagnosis. There exist approaches that detect problems in distributed systems by distributed checking of predicates. The online checker, D^3S, by Liu et al. [94] allows developers to specify predicates regarding the state of the distributed system under study. These are then checked

while the system is running. D3S uses binary instrumentation to avoid modifying the software and collects global snapshots of the state. While powerful, binary instrumentation adds non-trivial complexity and is limited in scale. Pip [124] is an example of another class of systems that trace the behavior of a running application and checks that behavior against programmer expectations and enables the programmer to then examine the resulting valid and invalid behavior.

Other systems aim at diagnosing problems in networks using inference, with data based on active probing [96] or passive monitoring [32]. Another class of systems aims at inferring causality based on collected message traces [16, 125]. They target the debugging and profiling of individual applications while our purpose is to support debugging of networks.

Pervasive tracing: Some proposals integrate improved in-band diagnosis and tracing support directly into the Internet. Fonseca et al. [67] proposes X-Trace, a pervasive cross-layer framework to enable in-band tracing of network problems. A trace ID is propagated from the application to the network stack and included in all associated messages and requests. Anand et al. [26] propose Net-Replay, a network primitive, which enables in-band tracing of network problems, in particular short-lived performance glitches, by marking and remembering recently seen packets throughout the Internet. Such approaches face deployment hurdles, due to the scale of changes involved. We focus on the more controllable environment of a single administrative domain, providing replay support directly in the substrate, with no changes required to the network.

Testing large-scale networks: Many approaches experience scalability issues when dealing with large networks. The authors of [78] suggest to scale down large networks and map them to smaller virtualized testbeds, combining *time dilation* [79] and disk I/O simulation to enable accurate behavior. This idea may aid scaling replay testbeds for **OFRewind**.

6.6 Summary

This work addresses an important void in debugging operational networks – scalable, economically feasible recording and replay capabilities. We present the design, implementation, and usage of **OFRewind**, a system capable of recording and replaying network events, motivated by our experiences troubleshooting network device and control plane anomalies. **OFRewind** provides control over the topology (choice of devices and their ports), timeline, and selection of traffic to be collected and then replayed in a particular debugging run. Using simple case studies, we highlight the potential of **OFRewind** for not only reproducing operational problems encountered

in a production deployment but also localizing the network events that trigger the error. According to our evaluation, the framework is lightweight enough to be enabled per default in production networks.

Some challenges associated with network replay are still under investigation, including *improved timing accuracy*, *multi-instance synchronization*, and *online replay*. **OFRewind** can preserve flow order, and its timing is accurate enough for many use cases. However, further improvements would widen its applicability. Furthermore, synchronization among multiple *Ofrecord* and *Ofreplay* instances is desirable, but nontrivial, and might require hardware support for accurate time-stamping [105].

Sometimes *Ofrecord* may only collect meta data but not the original data, e.g., only packet header. However, for reproducing the problem one may need the full set, e.g., full packets. Therefore, we plan to constructing dummy (synthetic) messages, e.g., packets with the same headers but random packet content.

In a possible extension of this work, *Ofrecord* and *Ofreplay* are combined to form an *online replay* mode. Recorded messages are directly replayed upon arrival, e.g., to a different set of hardware or to a different *substrate* slice. This may combine the multitude of operation modes and configuration features of **OFRewind** with the powerful online troubleshooting functionality of *Mirror VNets*.

Our next steps involve gaining further experience with more complex use cases. We plan to collect and maintain a standard set of traces that serve as input for automated regression tests, as well as benchmarks, for testing new network components. Thus, we expect **OFRewind** to play a major role in helping ongoing OpenFlow deployment projects[5] resolve production problems.

[5]There are ongoing production deployments of OpenFlow-enabled networks in Asia, Europe, as well as the US.

7

Conclusion and Outlook

Controlling and troubleshooting networks has been, is, and presumably will remain a challenging task. After all, the challenges intrinsic to networks, including the complexity of handling the distributed state and configuration, will not simply go away. Furthermore, we expect networks will continue to grow in speed, scale, and geographical spread. In particular, we foresee more and more traditionally local computing services move to distributed, cloud based infrastructures. Consequently, the availability and performance of such services depends on our ability to precisely control, instrument and troubleshoot the involved networks.

Indeed, the complexity of networks is increasing and will likely continue to increase fast. This is largely independent of any troubleshooting features, and driven by performance, flexibility, and cost requirements. Cloud services and Virtual Networks run on physical infrastructure shared by multiple, independent tenants. In such environments, troubleshooting a malfunction in an ad-hoc manner is very difficult for a tenant, as the lower layers of the architecture are inaccessible for inspection without dedicated troubleshooting interfaces. Software Defined Networks can forward based on arbitrary cross-layer flow attributes and thus greatly increase the flexibility for network operators. However, they also create new failure modes unknown in a classical routed network. Our approaches can serve as a guidance how we can "use virtualization to save it from itself", and leverage the additional power and flexibility incurred by virtual and software-defined networks to deal with their added complexity.

Reconsider our example from the introduction, where a residential user accesses an online spreadsheet served by a cloud service, and recall the chain of networks and connections that have to function:

1. the user's *residential home network* and *access line*
2. the *access network* of the user's Internet provider
3. the *inter-provider peerings* and the *core networks* of all involved transfer providers
4. the *datacenter network* in the service provider's DC
5. the actual *service* running on the server.

Due to the complexity, a one-size-fits-all solution for these different environments may be infeasible. Consequently, in this thesis, we propose and evaluate several approaches for these individual areas and scenarios. We now summarize our findings, then give directions for future work in the area.

7.1 Summary

In this thesis, we propose and evaluate the following approaches, architectures and systems:

Community Flow-Routing: We propose and evaluate *Flow-Routing* (Chapter 3) as an approach to improve reliability and performance of customer Internet access lines. Bandwidth utilization in residential networks is very uneven. While the average utilization remains low, the bandwidth may limit the quality of user experience in times of peak demand. This especially concerns large uploads, e.g., to social media sites, as the upstream bandwidth offered by typical consumer Internet providers is an order of magnitude below the marketed downstream bandwidth. The local access line is also a single point of failure for the user's Internet experience, as a failure of the local access line will prevent the user from accessing any cloud services at all. We find Flow-Routing to be a viable, cost effective approach for communities to share and bundle their access lines for improved reliability and performance, with an improvement of up to a factor of 3 in download times shown in our evaluation.

An Architecture For Network Virtualization: Virtual Networks may help improve the innovation speed of the Internet, and enable custom control and instrumentation abilities for services across providers. For instance, they enable a service provider to provide guaranteed *Quality of Experience* and customize routing and protocol stacks to optimize per application. We propose a *Control Architecture for Virtual Networks* (Chapter 4), the first such architecture that targets a multi-provider scenario. As such, our architecture emphasizes the

individual business roles and interfaces between the players involved, and also considers the business interests of the present stakeholders.

Mirror VNets: Based on Virtual Networks, we propose Mirror VNets (Chapter 5) which enable safe upgrades and troubleshooting in virtual networks, possibly spawning several independent infrastructures. To this end, a production VNet is paired with a *Mirror VNet* in identical state and configuration. Troubleshooting or upgrades can then be performed safely in the Mirror VNet, without affecting the production VNet. We present a prototype implementation and a case study in quality of experience scenario. We find that Mirror VNets work well if the underlying virtualization layer offers good *isolation*.

Control-Plane Record and Replay with OFRecord: We propose OFRewind (Chapter 6), the first system that leverages the flexibility of Software Defined Networks to enable practical Record and Relay troubleshooting in networks, even in the presence of *black-box devices* than cannot be modified or instrumented. We describe the design and implementation of the system, present several case studies that underline its utility for both OpenFlow and non-OpenFlow components, among them *switch (blackbox) malfunctions*, *controller* bugs and an *IP router problem*. We evaluate its scalability and find that OFRewind scales at least as well as current controller implementations and does not significantly impact the scalability of an OpenFlow controller domain.

In our work we prefer *network-centric* approaches. More specifically, we refrain from proposing pervasive approaches that modify many layers in the stack because of the associated deployment hurdles. In a similar vein, we emphasize approaches that work with unmodified end-hosts and software as far as possible. This enables our approaches to work in the presence of closed-source software and black-box devices—a reality in networks today.

In the spirit of proposing practically feasible solutions and keeping with the Internet credo of *running code*, our approaches have all been practically implemented "on the metal" in least in a prototypical fashion and evaluated on real networks[1]. *OFRewind* has spawned interest in the OpenFlow community as a production tool and may be extended to a production-ready open source project in the near future.

7.2 Future Directions

Gaining more experience with our approaches in larger scale deployments, such as the OpenFlow campus deployments in the US and Europe [4] as well as large-scale experimental networks such as GENI [70], is one of the main challenges for the near

[1]The VNet Control Architecture is being further refined, implemented and evaluated in ongoing, separate work.

future. As the deployments of the underlying enablers (Virtual Networks and Software Defined Networks) grow further beyond their current state, our troubleshooting and instrumentation approaches will have to grow with them. This growth will entail further scalability challenges, e.g., when OFRewind runs in a large-scale OpenFlow network controlled by a distributed controller. While we have presented an analysis of the theoretical scalability properties, we are convinced that further practical challenges will emerge once such deployments exist at scale.

There exist a multitude of trade-offs in the area of Software Defined Networks that influence our control and troubleshooting abilities and thus warrant investigation. For instance, flow *aggregation* has been suggested as a means to improve the scalability of OpenFlow domains. However, coarse-grained aggregated flow table entries may reduce our ability to selectively monitor, instrument, mirror and record traffic. So given a certain scenario, traffic mix, and instrumentation objectives, how fine-grained should the flow-definitions be? Which flows should be aggregated?

Distributed controllers have been suggested as another means to scale controller domains. Clearly, there exists a connection between the communication scheme of the distributed instances, the scalability of such distributed controllers, and the consistency properties they can achieve. The exact form that such controllers may take and consequently their impact on troubleshooting is not yet clear.

Lastly, there is a fundamental trade-off that affects all of our approaches: Any mechanism that adds flexibility for the purpose of powerful troubleshooting or customized control necessarily also adds complexity to the system itself, makes it more heavyweight and consequently prone to errors. The Internet has arguably fared very well by keeping the intelligence out of the network, its architecture simplistic and feature set minimal. However, there is increasing evidence that further growth in scale, performance and importance may not be sustainable based on these classical principles alone. Over-provisioning as the sole instrument of ensuring quality and reliability might not work in a world of streamed HD content and decreasing operator revenues—it is possible that the Internet may have to find a new *sweet spot* in this continuum.

Acknowledgments

This is the end of the thesis, the end of my time in Berlin and at INET. Time to say thank you to a lot of people, more than fit on a good page.

Not once during my diploma studies did I consider continuing for a Ph.D. and going into research. Now I am submitting my thesis and moving to California for a Post-Doc stipend—apparently, something in between must have changed my mind?!

It was, quite frankly, the atmosphere at INET, completely unlike anything I had seen at a German university before. Energetic, competitive, opinion-strong, always willing to take it out for the better arguments. Shooting high, playing with the big U.S. kids (and occasionally getting our backs kicked by them). A wholly nerve-wrecking and inspiring place to work.

The person to thank for this environment is, first and foremost, my advisor, Anja Feldmann, who is totally open about the *how?* and *where?*, but never fails to challenge and question the results. After all, in spite of appearances, she can always be persuaded by good arguments. And most importantly, she never fails to provide the crucial positive reinforcement, when another hard-worked submission comes back with the oh-so-well-known remark "We regret to inform you, but…". On my way, she taught me a hell of a lot, including but not limited to deciphering and decoding her 500-character long *perl* one-liners which are rightfully respected and feared throughout the galaxy. Anja, you set the bar very high for all the research `$bosses` that will come after you, as well as my own future contacts with students and interns, thanks a lot!

Of course, taking that journey into science alone would not have been quite so much fun. So a big thank you to my collaborators, co-authors and friends at INET! First up, Vlad Manilici, who got me into this mess in the first place, and routed a big number of flows with me—sadly, before the topic was hot. Anyway, we knew it back then! Merci to Olaf Maennel, Gregor Schaffrath, and Amir Mehmood, who took a stroll with me down Virtual Insanity Lane and shared the depths and shallows of a SIGCOMM hot topic, and the associated project and workshop experiences. Last but certainly not least in this list, Dan Levin, my faithful Routerlab *padawan* of many years and scientific collaborator of the last (and wildly most successful) year—thanks a lot for all the energy and sleep deprivation you invested into transforming

the OFRewind project from the mess it was at the beginning of 2010 to its current shiny state!

Also big cudos to Srini Seetharaman and Rob Sherwood, who hosted me during two extended research visits in the Golden State of California. They got me interested in the virtues and limits of Software Defined Networks, and convinced me of the benefits of Golden Sunsets, fresh avocados and incredible Sushi bars. Thanks a lot!

Then again, life with just work wouldn't rock so hard either[2]. I have had some great times on the road with my friends at INET. NYC 07 comes to mind (recall what conference it was, Thomas and Gregor?), a cool road trip down the pacific north west (Hey ho Microbrews, Fabian!), a crazy Barcelona stint with mostly everyone on board, hiking in Pyrenees (I say let's order the boar, Bernhard and Grsch!) and of course, great three months in CA with Nadi (and Dan the second time around). And, locally, countless parties in F'hain with the usual suspects (Messieurs TH and JSZ, I am looking at you), that made sure I will remember I was living "im Dicken B" and not in Gütersloh after all (well, as much as I remember those parties anyway, but that's another story and shall be told another time.) Cudos as well to everyone I have irresponsibly overlooked here!

Next I need to turn to my home-base in Geretsried that I haven't quite been able to abandon in spite of all the Big City allure. Big thanks to my crowd at teleteach, especially to Thomas Herrmann, founder, boss and fatherly friend, who always welcomed me and made do with the time I could spare for our little software house. Turning to the other brain half, greetings to my a-cappella band, Singsang, Baldi, Che and Kalle, who shared a similar fate of having to make do with whatever slots my work and travel schedules would leave open. Singing is an incredible hobby, a good song is a perfect shortcut to happiness and the world would be a better place if more people did it[3].

Lastly, the most important thank you goes to my family, especially to my parents, who always backed and supported me in the meandering paths between education, business and finally science I have chosen for myself. This is where I am now. Let's see how the path continues.

[2] even a researcher life, in spite of what the rumors say.
[3] Well and maybe *a few* people stopped...

List of Figures

List of Tables

Bibliography

[1] Annual Report Deutsche Telekom AG, 2008. http://www.download-telekom.de/dt/StaticPage/62/37/50/090227_DTAG_2008_Annual_report.pdf_623750.pdf.

[2] AT&T Reports Fourth-Quarter and Full-Year Results. http://www.att.com/gen/press-room?pid=4800&cdvn=news&newsarticleid=26597.

[3] AT&T Reports Fourth-Quarter and Full-Year Results. http://www.att.com/gen/press-room?pid=4800&cdvn=news&newsarticleid=26502.

[4] EU Project Ofelia. http://http://www.fp7-ofelia.eu/.

[5] IETF Working Group Forces. http://datatracker.ietf.org/wg/forces/charter/.

[6] NEC Programmable Networking Solutions. http://www.necam.com/PFlow/.

[7] Nicira Networks Inc. http://www.nicira.com/.

[8] OFRewind Code. www.openflow.org/wk/index.php/OFRewind.

[9] OpenVZ. http://wiki.openvz.org/.

[10] tcpdump. http://www.tcpdump.org/.

[11] tcpreplay. http://tcpreplay.synfin.net/.

[12] VINI End User Attachment. http://www.vini-veritas.net/documentation/pl-vini/user/clients.

[13] Vmware infrastructure. http://www.vmware.com/products/vi/.

[14] CIO update: Post-mortem on the Skype outage. http://blogs.skype.com/en/2010/12/cio_update.html, December 29 2010.

[15] 4WARD Project. http://www.4ward-project.eu.

[16] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance debugging for distributed systems of black boxes. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2003), SOSP '03, pp. 74–89.

[17] Akella, A., Maggs, B., Seshan, S., Shaikh, A., and Sitaraman, R. K. A measurement-based analysis of multihoming. In *Proceedings of ACM SIGCOMM conference on Data communication* (2003), pp. 353–364.

[18] Akella, A., Seshan, S., and Shaikh, A. Multihoming performance benefits: An experimental evaluation of practical enterprise strategies. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2004).

[19] Alimi, R., Wang, Y., and Yang, Y. R. Shadow configuration as a network management primitive. In *Proceedings of ACM SIGCOMM conference on Data communication* (2008).

[20] Altekar, G., and Stoica, I. ODR: output-deterministic replay for multicore debugging. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2009), SOSP '09.

[21] Altekar, G., and Stoica, I. Dcr: Replay debugging for the datacenter. Tech. Rep. UCB/EECS-2010-74, UC Berkeley, 2010.

[22] Altekar, G., and Stoica, I. Focus replay debugging effort on the control plane. In *Proc. USENIX HotDep* (2010), HotDep'10, pp. 1–9.

[23] Amazon Inc. Amazon Web Services. `http://aws.amazon.com/`.

[24] Amazon Inc. Summary of the Amazon EC2 and Amazon RDS Service Disruption in the US East Region. `http://aws.amazon.com/message/65648/`, May 25 2011.

[25] Anagran. `http://www.anagran.com`.

[26] Anand, A., and Akella, A. Netreplay: a new network primitive. In *Proc. HOTMETRICS* (2009).

[27] Anderson, E., and Arlitt, M. Full Packet Capture and Offline Analysis on 1 and 10 Gb/s Networks. Tech. Rep. HPL-2006-156, HP Labs, 2006.

[28] Anderson, T., Peterson, L., Shenker, S., and Turner, J. Overcoming the internet impasse through virtualization. *IEEE Computer Magazine 38*, 4 (2005).

[29] Antonelli, C., Co, K., m Fields, and Honeyman, P. Cryptographic Wiretapping at 100 Megabits. In *16th International Symposium on Aerospace Defense Sensing, Simulation, and Controls (SPIE)* (2002).

[30] Anwer, M. B., Nayak, A., Feamster, N., and Liu, L. Network I/O fairness in virtual machines. In *Proceedings of the ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)* (2010), pp. 73–80.

[31] Bagnulo, M., and Nordmark, E. Shim6: Level 3 Multihoming Shim Protocol for IPv6. RFC 5533 (Standards track), 2009.

[32] Bahl, P., Chandra, R., Greenberg, A., Kandula, S., Maltz, D. A., and Zhang, M. Towards highly reliable enterprise network services via inference of multilevel dependencies. In *Proceedings of ACM SIGCOMM conference on Data communication* (New York, NY, USA, 2007), SIGCOMM '07, ACM, pp. 13–24.

[33] Ballard, J. R., Rae, I., and Akella, A. Extensible and scalable network monitoring using opensafe. In *Proceedings of the USENIX Internet Management Workshop/Workshop on Research on Enterprise Networking (INM/WREN)* (Berkeley, CA, USA, 2010), USENIX Association, pp. 8–8.

[34] Barford, P., and Crovella, M. Generating representative web workloads for network and server performance evaluation. *SIGMETRICS Performance Evaluation Review 26* (June 1998), 151–160.

[35] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. XEN and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2003), pp. 164–177.

[36] Bavier, A., Bowman, M., Chun, B., Culler, D., Karlin, S., Muir, S., Peterson, L., Roscoe, T., Spalink, T., and Wawrzoniak, M. Operating system support for planetary-scale network services. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2004), pp. 19–19.

[37] Beacon—A Java-based OpenFlow controller. `http://www.openflowhub.org/display/Beacon/Beacon+Home`.

[38] Bhatia, S., Motiwala, M., Mühlbauer, W., Mundad, Y., Valancius, V., Bavier, A., Feamster, N., Peterson, L., and Rexford, J. Trellis: A platform for building flexible, fast virtual networks on commodity hardware. In *Proceedings of the 3rd International Workshop on Real Overlays And Distributed Systems (ACM ROADS)* (2008).

[39] Bienkowski, M., Feldmann, A., Jurca, D., Kellerer, W., Schaffrath, G., Schmid, S., and Widmer, J. Competitive analysis for service migration in vnets. In *Proceedings of the ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)* (2010), VISA '10, pp. 17–24.

[40] Blackford, R. Try the blue pill: What's wrong with life in a simulation? In *Jacking in to the Matrix franchise: cultural reception and interpretation* (New York, NY, 2004), M. Kapell and W. G. Doty, Eds., Continuum International Publishing, pp. 169–171.

[41] Bychkovsky, V., Hull, B., Miu, A. K., Balakrishnan, H., and Madden, S. A Measurement Study of Vehicular Internet Access Using In Situ Wi-Fi Networks. In *MOBICOM* (2006).

[42] CBench - Controller Benchmarker. `www.openflowswitch.org/wk/index.php/Oflops`.

[43] Chandra, R., Bahl, V., and Bahl, P. Multinet: Connecting to multiple IEEE 802.11 networks using a single wireless card. In *Proceedings of the IEEE Annual International Conference on Computer Communications (INFOCOM)* (2004).

[44] Chandrasekaran, S., and Franklin, M. Remembrance of Streams Past: Overload-sensitive Management of Archived Streams. In *Proceedings of the Thirtieth international conference on Very large data bases (VLDB)* (2004).

[45] Chowdhury, N., Rahman, M., and Boutaba, R. Virtual network embedding with coordinated node and link mapping. In *INFOCOM 2009, IEEE* (april 2009), pp. 783 –791.

[46] Cisco Inc. Secure Domain Router Commands on Cisco IOS XR Software. `http://www.cisco.com/en/US/docs/ios_xr_sw/iosxr_r3.4/system_management/command/reference/yr34sdr.html`.

[47] Clark, C., Fraser, K., H, S., Hansen, J. G., Jul, E., Limpach, C., Pratt, I., and Warfield, A. Live Migration of Virtual Machines. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2005), pp. 273–286.

[48] Clark,, D. D., Wroclawski,, J., Sollins,, K. R., and Braden,, R. Tussle in Cyberspace: Defining Tomorrow's Internet. In *Proceedings of ACM SIGCOMM conference on Data communication* (2002), pp. 347–356.

[49] Cooke, E., Myrick, A., Rusek, D., and Jahanian, F. Resource-aware Multi-format Network Security Data Storage. In *Proceedings of the ACM SIGCOMM Workshop on Large Scale Attack Defense (LSAD)* (2006), pp. 177–184.

[50] Crovella, M. E., and Bestavros, A. Self-similarity in World Wide Web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking 5* (December 1997), 835–846.

[51] Dai, R., Stahl, D. O., and Whinston, A. B. The economics of smart routing and quality of service. In *Group Communications and Charges. Technology and Business Models* (2003), vol. 2816, pp. 318–331.

[52] Desnoyers, P., and Shenoy, P. J. Hyperion: High Volume Stream Archival for Retrospective Querying. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2007), pp. 45–58.

[53] Dike, J. A usermode port for the linux kernel. In *USENIX Linux Showcase & Conference* (2000).

[54] Dobrescu, M., Argyraki, K., Iannaccone, G., Manesh, M., and Ratnasamy, S. Controlling parallelism in a multicore software router. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (ACM PRESTO)* (2010), pp. 2:1–2:6.

[55] DropBox file sychronization service. `http://www.dropbox.com/`.

[56] Dunlap, G. W., King, S. T., Cinar, S., Basrai, M. A., and Chen, P. M. Revirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2002).

[57] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., and Mathy, L. Fairness issues in software virtual routers. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (ACM PRESTO)* (2008), pp. 33–38.

[58] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Huici, F., Mathy, L., and Papadimitriou, P. Forwarding path architectures for multicore software routers. In *Proceedings of the Workshop on Programmable Routers for Extensible Services of Tomorrow (ACM PRESTO)* (2010), PRESTO '10, pp. 3:1–3:6.

[59] Egi, N., Greenhalgh, A., Handley, M., Hoerdt, M., Mathy, L., and Schooley, T. Evaluating XEN for router virtualization. In *Proceedings of the IEEE International Workshop on Performance Modeling and Evaluation of Computer and Telecommunication Networks (PMECT)* (2007).

[60] The E-model, a Computational Model for Use in Transmission Planning, ITU-T Rec. G.107, 2005.

[61] Fang, L., Bita, N., Le Roux, J.-L., and Miles, J. Interprovider IP-MPLS services: requirements, implementations, and challenges. *IEEE Communication Magazine 43*, 6 (june 2005), 119 – 128.

[62] Feamster, N., and Balakrishnan, H. Detecting bgp configuration faults with static analysis. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (Berkeley, CA, USA, 2005), USENIX Association, pp. 43–56.

[63] Feamster, N., Gao, L., and Rexford, J. How to Lease the Internet in Your Spare Time. *ACM SIGCOMM Computer Communication Review (CCR) 37*, 1 (2007), 61–64.

[64] Feldmann, A. Internet clean-slate design: What and why? *ACM SIGCOMM Computer Communication Review (CCR) 37*, 3 (2007), 59–64.

[65] Feldmann, A., Kind, M., Maennel, O., Schaffrath, G., and Werle, C. Network Virtualization – An Enabler for Overcoming Ossification. *ERCIM News 77* (April 2009), 21—23.

[66] Field, T. Facebook revolution: Hold the hyperbole. `http://www.thespec.com/opinion/article/510112`, 2011.

[67] Fonseca, R., Porter, G., Katz, R. H., Shenker, S., and Stoica, I. X-trace: A pervasive network tracing framework. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2007).

[68] Geels, D., Altekar, G., Maniatis, P., Roscoe, T., and Stoica, I. Friday: Global comprehension for distributed replay. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2007).

[69] Geels, D., Altekar, G., Shenker, S., and Stoica, I. Replay debugging for distributed applicatinos. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2006).

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

[71] ProtoGENI: RSpec. `http://www.protogeni.net/trac/protogeni/wiki/RSpec`.

[72] Goldenberg, D., Qiu, L., Xie, H., Yang, Y. R., and Zhang, Y. Optimizing cost and performance for multihoming. In *Proceedings of ACM SIGCOMM conference on Data communication* (2004).

[73] Goldenberg, D. K., Qiu, L., Xie, H., Yang, Y. R., and Zhang, Y. Optimizing cost and performance for multihoming. In *Proceedings of ACM SIGCOMM conference on Data communication* (2004).

[74] Gonzalez, J. M., Paxson, V., and Weaver, N. Shunting: A Hardware/Software Architecture for Flexible, High-performance Network Intrusion Prevention. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)* (2007), pp. 139–149.

[75] Google Inc. Google Docs. `https://docs.google.com`, 2011.

[76] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: Towards an operating system for networks. *ACM SIGCOMM Computer Communication Review (CCR) 38*, 3 (2008), 105–110.

[77] Guo, F., Chen, J., Li, W., and cker Chiueh, T. Experiences in building A multihoming load balancing system. In *Proceedings of the IEEE Annual International Conference on Computer Communications (INFOCOM)* (2004).

[78] Gupta, D., Vishwanath, K., and Vahdat, A. Diecast: Testing distributed systems with an accurate scale model. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2008).

[79] Gupta, D., Yocum, K., Mcnett, M., Snoeren, A. C., Vahdat, A., and Voelker, G. M. To infinity and beyond: Time warped network emulation. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)* (2005).

[80] Habib, A., Christin, A., and Chuang, J. Taking Advantage of Multihoming with Session Layer Striping. In *IEEE Global Internet* (2006).

[81] Hickey, A. R. Juniper opens os to third-party developers, taking stab at cisco. `http://www.crn.com/news/networking/204800583/juniper-opens-os-to-third-party-developers-taking-stab-at-cisco.htm`.

[82] Hower, D. R., Montesinos, P., Ceze, L., Hill, M. D., and Torrellas, J. Two hardware-based approaches for deterministic multiprocessor replay. *Communications of the ACM 52*, 6 (2009), 93–100.

[83] Hsieh, H.-Y., and Sivakumar, R. A transport layer approach for achieving aggregate bandwidths on multi-homed mobile hosts. In *Proceedings of the ACM Annual International Conference on Mobile Computing and Networking (MobiCom)* (2002).

[84] Intel Virtualization Technology: Hardware support for efficient processor virtualization. `http://www.intel.com/technology/itj/2006/v10i3/1-hardware/6-vt-x-vt-i-solutions.htm`.

[85] Internap. `http://www.internap.com`.

[86] Juniper Inc. Control Plane Scaling and Router Virtualization. `www.juniper.net/us/en/local/pdf/whitepapers/2000261-en.pdf`.

[87] Keller, E., Yu, M., Caesar, M., and Rexford, J. Virtually eliminating router bugs. In *Proceedings of the ACM International Conference on emerging Networking EXperiments and Technologies (CoNEXT)* (2009).

[88] Kim, C., Caesar, M., and Rexford, J. Floodless in Seattle: A scalable Ethernet architecture for large enterprises. *ACM SIGCOMM Computer Communication Review (CCR) 38* (August 2008), 3–14.

[89] King, S. T., Dunlap, G. W., and Chen, P. M. Debugging operating system with time-traveling virtual machines. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)* (2005).

[90] Kivity, A., Kamay, Y., Laor, D., Lublin, U., and Liguori, A. KVM: the Linux virtual machine monitor. In *Prooceedings of the Linux Symposium* (2007).

[91] Lad, M., Bhatti, S., Hailes, S., and Kirstein, P. Coalition-Based Peering for Flexible Connectivity. In *Proceedings of the Annual IEEE International Symposium on Personal, Indoor and Mobile Radio Communication (PIMRC)* (2006).

[92] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM 21*, 7 (1978).

[93] Lin, C.-C., Caesar, M., and van der Merwe, J. Towards interactive debugging for ISP networks. In *ACM HotNets Workshop* (2009).

[94] Liu, X., Guo, Z., Wang, X., Chen, F., Lian, X., Tang, J., Wu, M., Kaashoek, M. F., and Zhang, Z. D3S: debugging deployed distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2008), pp. 423–437.

[95] Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., and Luo, J. Netfpga–an open platform for gigabit-rate network switching and routing. In *Proceedings of the 2007 IEEE International Conference on Microelectronic Systems Education* (2007), MSE '07, pp. 160–161.

[96] Mahajan, R., Spring, N., Wetherall, D., and Anderson, T. User-level internet path diagnosis. *ACM SIGOPS Operating System Review 37*, 5 (2003), 106–119.

[97] Mahajan, R., Wetherall, D., and Anderson, T. Understanding BGP misconfiguration. In *Proceedings of ACM SIGCOMM conference on Data communication* (2002), pp. 3–16.

[98] Maier, G., Feldmann, A., Paxson, V., and Allman, M. On dominant characteristics of residential broadband internet traffic. In *ACM IMC* (2009), pp. 90–102.

[99] Maier, G., Sommer, R., Dreger, H., Feldmann, A., Paxson, V., and Schneider, F. Enriching network security analysis with time travel. In *Proceedings of ACM SIGCOMM conference on Data communication* (2008).

[100] Malik, O. The Telia-Cogent spat could ruin the web for many. `http://gigaom.com/2008/03/14/the-telia-cogent-spat-could-ruin-web-for-many/`.

[101] Manilici, V., and Wundsam, A. FlowSim – a flow based network simulator. `http://www.net.t-labs.tu-berlin.de/~vlad/FlowRoutingSoftware`, 2008.

[102] Mansmann, U. Beschwerdeflut: Was bei DSL alles schiefgehen kann. *c't: Magazin für Computer-Technik 9* (2009), 152–154.

[103] McGrath, K. P., and Nelson, J. Monitoring & Forensic Analysis for Wireless Networks. In *Proceedings of the IEEE Conference on Internet Surveillance and Protection (ICISP)* (2006).

[104] McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., and Turner, J. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review (CCR) 38*, 2 (2008).

[105] Micheel, J., Donnelly, S., and Graham, I. Precision timestamping of network packets. In *Proceedings of the ACM Internet Measurement Workshop (IMW)* (2001).

[106] Montesinos, P., Hicks, M., King, S. T., and Torrellas, J. Capo: a software-hardware interface for practical deterministic multiprocessor replay. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (2009), pp. 73–84.

[107] Mushroom Networks. http://www.mushroomnetworks.com.

[108] Nichols, K., Blake, S., Baker, F., and Black, D. Definition of the differentiated services field (ds field) in the ipv4 and ipv6 headers. RFC 2474 (Standards track), 1998.

[109] Nistnet. http://www-x.antd.nist.gov/nistnet.

[110] NOX - An OpenFlow Controller. www.noxrepo.org.

[111] Odlyzko, A. M. Data networks are lightly utilized, and will stay that way. *Review of Network Economics* (2003).

[112] Open Networking Foundation. http://www.opennetworkingfoundation.org/.

[113] OpenVSwitch. http://http://openvswitch.org/.

[114] Papadopouli, M., and Schulzrinne, H. Connection sharing in an ad hoc wireless network among collaborating hosts. In *Proceedings of the ACM Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)* (1999).

[115] Pfaff, B., Pettit, J., Amidon, K., Casado, M., Koponen, T., and Shenker, S. Extending networking into the virtualization layer. In *ACM HotNets Workshop* (2009), New York.

[116] Phaal, P., Panchen, S., and McKee, N. Inmon corporation's sflow: A method for monitoring traffic in switched and routed networks. RFC 3176, 2001.

[117] Open Source SIP Stack and Media Stack for Presence, Instant Messaging, and Multimedia Communication, 2009. http://www.pjsip.org.

[118] The 16-bit AS Number Report. http://www.potaroo.net/tools/asn16/.

[119] Quagga Routing Suite. www.quagga.net.

[120] Radware. http://www.radware.com.

[121] Raza, S., Huang, G., Chuah, C.-N., Seetharaman, S., and Singh, J. MeasuRouting: A Framework for Routing Assisted Traffic Monitoring. In *Proceedings of the IEEE Annual International Conference on Computer Communications (INFOCOM)* (March 2010), pp. 1–9.

[122] Reddit: The Voice of the Internet. http://www.reddit.com/.

[123] Reiss, F., Stockinger, K., Wu, K., Shoshani, A., and Hellerstein, J. M. Enabling real-time querying of live and historical stream data. In *Proceedings of the 19th International Conference on Scientific and Statistical Database Management* (2007), SSDBM '07.

[124] Reynolds, P., Killian, C., Wiener, J. L., Mogul, J. C., Shah, M. A., and Vahdat, A. Pip: Detecting the unexpected in distributed systems. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2006).

[125] Reynolds, P., Wiener, J. L., Mogul, J. C., Aguilera, M. K., and Vahdat, A. WAP5: black-box performance debugging for wide-area systems. In *Proceedings of the International World Wide Web Conference (WWW)* (New York, NY, USA, 2006), ACM, pp. 347–356.

[126] RIPE. YouTube Hijacking: A RIPE NCC RIS case study . http://www.ripe.net/internet-coordination/news/industry-developments/youtube-hijacking-a-ripe-ncc-ris-case-study, March 2008.

[127] Rivas, R., Arefin, A., and Nahrstedt, K. Janus: a cross-layer soft real-time architecture for virtualization. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing* (New York, NY, USA, 2010), HPDC '10, ACM, pp. 676–683.

[128] Robertson, S. P., Vatrapu, R. K., and Medina, R. The social life of social networks: Facebook linkage patterns in the 2008 u.s. presidential election. In *Proceedings of the Annual International Conference on Digital Government Research: Social Networks: Making Connections between Citizens, Data and Government* (2009), dg.o '09, pp. 6–15.

[129] Rodriguez, P., Chakravorty, R., Chesterfield, J., Pratt, I., and Banerjee, S. MAR: a commuter router infrastructure for the mobile internet. In *Proceedings of the The International Conference on Mobile Systems, Applications, and Services (MobiSys)* (2004).

[130] Roe, C., and Gonik, S. Server-side design principles for scalable Internet systems. *IEEE Software 19*, 2 (mar/apr 2002), 34 –41.

[131] Russell, A. 'rough consensus and running code' and the internet-osi standards war. *Annals of the History of Computing, IEEE 28*, 3 (july-sept. 2006), 48 –61.

[132] Schulz-Zander, J., Zhu, J., and Yiakoumis, Y. OpenFlow 1.0 for OpenWRT. http://www.openflow.org/wk/index.php/OpenFlow_1.0_for_OpenWRT.

[133] Shanmugasundaram, K., Memon, N., Savant, A., and Brönnimann, H. ForNet: A Distributed Forensics Network. In *Proceedings of the Workshop on Mathematical Methods, Models and Architectures for Computer Networks Security* (2003).

[134] Sherwood, R. An Experimenter's Guide to OpenFlow. In *GENI Experimenters Workshop* (2010).

[135] Sherwood, R., Chan, M., Covington, G. A., Gibb, G., Flajslik, M., Handigol, N., Huang, T.-Y., Kazemian, P., Kobayashi, M., Naous, J., Seetharaman, S., Underhill, D., Yabe, T., Yap, K.-K., Yiakoumis, Y., Zeng, H., Appenzeller, G., Johari, R., McKeown, N., and Parulkar, G. M. Carving research slices out of your production networks with openflow. *ACM SIGCOMM Computer Communication Review (CCR)* (2010), 129–130.

[136] Sherwood, R., Gibb, G., Yap, K.-K., Appenzeller, G., Casado, M., McKeown, N., and Parulkar, G. Can the production network be the testbed? In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)* (2010), pp. 1–6.

[137] Private communication with a Small Bavarian Internet Provider(TM).

[138] Sommers, J., and Barford, P. Self-configuring network traffic generation. In *ACM IMC* (2004).

[139] Sommers, J., Barford, P., Duffield, N., and Ron, A. Improving accuracy in end-to-end packet loss measurement. In *Proceedings of ACM SIGCOMM conference on Data communication* (2005), pp. 157–168.

[140] Stanford University Clean Slate Lab . http://cleanslate.stanford.edu/.

[141] Stewart, R. Stream Control Transmission Protocol. RFC 4960 (Proposed Standard), Sept. 2007.

[142] Stewart, R., and Metz, C. SCTP: new transport protocol for TCP/IP. *IEEE Internet Computing* (2001).

[143] Stone, B., and Cohen, N. Social networks spread defiance online. New York Times, June 16, 2009, page A11. http://www.nytimes.com/2009/06/16/world/middleeast/16media.html, June 16 2009.

[144] Crossbow: Network Virtualization and Resource Control. http://hub.opensolaris.org/bin/view/Project+crossbow/WebHome.

[145] Tao, S., Xu, K., Xu, Y., Fei, T., Gao, L., Guerin, R., Kurose, J., Towsley, D., and Zhang, Z.-L. Exploring the performance benefits of end-to-end path switching. In *Proceedings of the IEEE International Conference on Network Protocols (ICNP)* (2004).

[146] Private communication with a Large European Internet Service Provider(TM).

[147] Thompson, N., He, G., and Luo, H. Flow Scheduling for End-host Multihoming. In *IEEE Infocom* (2006).

[148] Tootoonchian, A., Ghobadi, M., and Ganjali, Y. OpenTM: traffic matrix estimator for OpenFlow networks. In *Proceedings of the Passive and Active Measurement Conference (PAM)* (2010), pp. 201–210.

[149] Urs Hoelzle. This is your pilot speaking. Now, about that holding pattern... http://googleblog.blogspot.com/2009/05/this-is-your-pilot-speaking-now-about.html, May 14 2009.

[150] Vini. http://www.vini-veritas.net/.

[151] viprinet. http://www.viprinet.com.

[152] Virtual Box. http://www.virtualbox.org/.

[153] Linux VServer. http://linux-vserver.org/.

[154] Wang,, Y., Keller,, E., Biskeborn,, B., van der Merwe,, J., and Rexford,, J. Virtual Routers on the Move: Live Router Migration as a Network-Management Primitive. *ACM SIGCOMM Computer Communication Review (CCR) 38*, 4 (2008), 231–242.

[155] WDS Linked Router Network. http://www.dd-wrt.com/wiki/index.php/WDS_Linked_router_network.

[156] White, B., Lepreau, J., Stoller, L., Ricci, R., Guruprasad, S., Newbold, M., Hibler, M., Barb, C., and Joglekar, A. An integrated experimental environment for distributed systems and networks. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[157] Wundsam, A. Connection sharing in community networks - how to accomodate peak bandwidth demands. Diplomarbeit, TU Munich, Germany, Apr. 2007.

[158] Yan, H., Maltz, D. A., Ng, T. S. E., Gogineni, H., Zhang, H., and Cai, Z. Tesseract: A 4D network control plane. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)* (2007).

[159] Yeow, W.-L., Westphal, C., and Kozat, U. Designing and embedding reliable virtual infrastructures. In *Proceedings of the ACM SIGCOMM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)* (2010), pp. 33–40.

[160] Yu, M., Yi, Y., Rexford, J., and Chiang, M. Rethinking virtual network embedding: substrate support for path splitting and migration. *ACM SIGCOMM Computer Communication Review (CCR) 38* (March 2008), 17–29.

[161] Zhu, Y., Zhang-Shen, R., Rangarajan, S., and Rexford, J. Cabernet: Connectivity architecture for better network services. In *Proceedings of the ACM Re-Architecting the Internet Workshop (ReArch)* (2008).