TRUSTWORTHY KNOWLEDGE PLANES FOR FEDERATED DISTRIBUTED SYSTEMS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Alan Chi-an Shieh

May 2012

TRUSTWORTHY KNOWLEDGE PLANES FOR FEDERATED DISTRIBUTED SYSTEMS

Alan Chi-an Shieh, Ph.D.

Cornell University 2012

In federated distributed systems, such as the Internet and the public cloud, the constituent systems can differ in their configuration and provisioning, resulting in significant impacts on the performance, robustness, and security of applications. Yet these systems lack support for distinguishing such characteristics, resulting in uninformed service selection and poor inter-operator coordination.

This thesis presents the design and implementation of a trustworthy knowledge plane that can determine such characteristics about autonomous networks on the Internet. A knowledge plane collects the state of network devices and participants. Using this state, applications infer whether a network possesses some characteristic of interest. The knowledge plane uses attestation to attribute state descriptions to the principals that generated them, thereby making the results of inference more trustworthy. Trustworthy knowledge planes enable applications to establish stronger assumptions about their network operating environment, resulting in improved robustness and reduced deployment barriers.

We have prototyped the knowledge plane and associated devices. Experience with deploying analyses over production networks demonstrate that knowledge planes impose low cost and can scale to support Internet-scale networks.

BIOGRAPHICAL SKETCH

Alan Shieh was born in Taiwan and immigrated to the United States during early childhood. He graduated with high honors from the University of California, Berkeley with a Bachelor's degree in electrical engineering and computer science, having specialized in computer science and fulfilled his honors breadth with coursework in mathematics. He is a member of the Eta Kappa Nu and Tau Beta Pi honor societies and was partially supported in his graduate studies by a Cornell University Fellowship.

ACKNOWLEDGMENTS

Doctoral study is often compared to a marathon. True to that analogy, I have received invaluable support from many coaches, teammates, friends, and family.

I would like to thank my advisor, Prof. Emin Gün Sirer. His tireless efforts helped me to develop a taste for recognizing, motivating, and solving novel problems in computer science. Profs. Andrew C. Myers and Fred B. Schneider have also guided my research over the years. The demand for precision by the faculty at Cornell taught me how to think and communicate clearly. Thanks to these lessons, I now appreciate that having a novel idea for solving a technical challenge is just a small step in advancing our knowledge of software systems. Creating a unit of knowledge in this domain is best done by realizing an idea in a representative prototype, then distilling and disseminating the lessons from this experience. Compared to the input idea, the output ideas from this process are substantially better proven and more logically coherent.

I would also like to thank Dr. Srikanth Kandula for supervising my internship at Microsoft Research, Redmond. Thanks to that experience, I gained invaluable insights into the interplay of research and industry both at the focused level of a particular research project and at an institutional level.

Intellectual labor can only be sustained by a supportive culture and community. The Systems lab at Cornell was filled with many friendly faces that brought vitality and good spirits to a sea of beige furniture and computer equipment. Our time working there gave rise to many friendships, and I will look back fondly at our time together at Cornell.

Too often, the student experience in a sleepy college town such as Ithaca is is defined by relationships between students who are temporarily stopping through. Thus, more than any other period in one's life, the sense of home and belonging is localized much more to a specific time than it is to a location. So I am thankful to have made dear friends, such as Elizabeth Kluz and Rebecca Plante, that call Ithaca their home. Through them, I can continue to consider Ithaca, the location, one of my homes as well.

Finally, I would like to thank my parents, Frederick and Shelly, and brother Peter for providing me with a supportive environment and for encouraging my academic pursuits.

Technical acknowledgments

This dissertation relies on several key technical contributions from my collaborators.

- The Nexus group developed attestation abstractions at the operating systems level that inspired the attribution-based knowledge plane used in NetQuery. The group also developed system infrastructure used by many NetQuery applications.
- Dr. Oliver Kennedy implemented substantial portions of the NetQuery tuplestore during his doctoral studies at Cornell.
- Dr. Srikanth Kandula contributed the motivating study identifying performance interference in operational infrastructure networks (6.1) and developed the stability analysis of the Seawall control loop (6.2).

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

This dissertation examines properties and claims. Properties are the known facts associated with a system, such as a network or computational cloud, ranging from ground truths about an individual device (e.g., router, switch, or end host), to logically inferred conclusions about an interconnected collection of devices (e.g., subnet, autonomous system, or data center). A claim is a special type of property that embodies a promise that a system meets particular requirements, such as performance, reliability, or integrity, of users and applications. Providing a ubiquitous interface for querying a system's properties and claims can simplify applications and streamline coordination between different entities in a system. However, due to complexity, cost, and confidentiality concerns, existing systems provide few mechanisms for determining whether a certain claim holds or for enforcing a claim.

This thesis examines new inter-domain coordination mechanisms for querying properties and checking claims in federated distributed systems built from many independently-operated participants. The overarching framework is that of a *knowledge plane* [46], called *NetQuery*, which collects and disseminates properties to enable application reasoning about the system.

In NetQuery, applications use *analysis* to infer some high-level characteristics of interest from ground properties of the system. An analysis is any program that takes ground properties as inputs and outputs a statement affirming some characteristic. For instance, to affirm that a network has reserved bandwidth between two hosts, an analysis could continuously monitor the configuration of intervening routers to verify that they provide a minimum bandwidth.

Compared to existing techniques for extracting characteristics, analysis can infer characteristics even when they are difficult or impossible to ascertain through measurements from an application's vantage point. For instance, failure resilience properties of networks are a critical differentiator between service providers. Yet it is difficult to verify the presence of backup paths through application-level probing. By comparison, expected performance and fault-tolerance under failure can be inferred from topology, link capacity, and failover configuration.

Reasoning is only meaningful if there is a basis for trusting the information contained in properties, since analysis driven by false information could lead to incorrect operation. For instance, an unscrupulous operator that operates a poorly-constructed network, yet exports properties matching those of a high-reliability network might fool applications into choosing their network. The operator thus collects an undeserved premium for service while degrading the application.

Such concerns arise because federated systems are diverse: participants are not always incentivized to be honest and hence do not always trust one another. To enable application reasoning to manage such differences, NetQuery records *attributions* that bind each property to the principal making that claim. By making this attribution metadata available to reasoning, applications can decide whether to use a property based on whether it comes from a trustworthy participant.

Attribution enables applications to make such decisions based on flexible, application-specific policies. This thesis describes how to use secure hardware coprocessors and software stacks for trustworthy computing to scalably and inexpensively generate such credentials while maintaining high assurance. Hardware coprocessors, such as the trusted platform module (TPM) [56], are cheap and becoming ubiquitous. TPMs are used to implement several important knowledge plane abstractions: they can be used as (1) the root of trust for attribution metadata, (2) to disseminate properties and make inferences in a manner that conforms to the often restrictive information dissemination policies of each administrative domain, and (3) to deploy system monitoring mechanisms that verify the accuracy of exported properties. Together, these mechanisms can be composed to show the validity of claims covering a whole system.

This thesis also introduces policy enforcement and resource scheduling mechanisms that provide guarantees about system behavior that are not supported by current networks. These new mechanisms expand the set of characteristics and claims that a network can support and can be implemented at low cost and high performance with only a small trusted computing base, lowering the cost of audit and strengthening the attack model to cover a wide range of adversaries.

1.1 Using analysis to support inter-domain coordination

Several applications of analysis to inter-domain coordination highlight the value of knowledge planes. These applications use analysis to check mutual requirements in network-to-host, host-to-network, and network-to-network interactions that are commonly found in federated networks such as the Internet.

- **Network-to-host.** Many networks employ access control policies to exclude potentially disruptive entities. In doing so, such policies provide an extra layer of defense against attack. For instance, a host running unpatched software can serve as a foothold into a network for an attacker. Suppose each host exported its process list as a property. Then a network can check the process list property of each host against a whitelist, admitting the host only if every process corresponds to an approved binary.

- **Host-to-network.** Mobile users often connect only to wireless hotspots that meet some expected level of integrity and performance. This policy protects against man-in-the-middle attacks originating from within the hotspot infrastructure and ensures good user experience. Suppose the devices composing a hotspot exported properties describing their platform (i.e., hardware and software) and topology. Then a mobile user can directly establish that a network has integrity using a structural check, verifying that every component used to implement the hotspot is a well-known access point, bridge, or gateway device. With minor changes, structural checks can establish the expected performance levels for all access points within range, enabling users to pick the best access point to use. By contrast, users today typically connect only to hotspots matching the identity of a known operator, e.g., with key-based authentication [4], with the implicit assumption that an authenticated network has the desired integrity or performance.

- **Network-to-network.** Many network relationships, such as those between providers and customers, are governed by contractual obligations. Such contracts on the Internet include peering agreements, which typically impose reliability requirements on the topology of each participant [18]; mutual backup agreements, which impose restrictions on when backup paths are allowed to be used; and service level agreements (SLAs), which promise bounds on performance and reliability during nominal and degraded network conditions. Compliance can be verified by checking whether the devices and interconnect are configured to deliver the promised performance.

1.2 Thesis scope and methodology

This thesis describes the design and implementation of a knowledge plane for federated distributed systems. This work targets the networking and cloud computing domains and shows the benefits and implementation considerations of adding a knowledge plane to the Internet and multi-tenant cloud datacenters.

We take advantage of several common architectural features of these federated systems in the design and implementation of the knowledge plane and associated applications. The infrastructure interfaces for system management, configuration, and data processing in such systems are particularly well-suited for supporting analysis:

- The system management interfaces (e.g., SNMP and CIM [35, 3]) export standardized, extensive descriptions of system state. The wealth of ground properties that these interfaces already collect can contribute greatly to the expressiveness of analysis since they provide detailed coverage of the vast majority of system components.
- Analysis is more tractable to perform with the control and data processing interfaces of these systems, since much of the control and data plane behavior is expressed in domain-specific languages (e.g., switch ACLs, forwarding tables, and configurations [61] for networks; bandwidth, memory, CPU allocations and VM containers [8] for clouds).

- Contractual stipulations concisely define the system requirements expected by users and applications. Contracts can serve as the basis for identifying and specifying characteristics of interest and in extracting information sharing policies.

As a result of these architectural features, only a few control- and data-plane modifications are needed to support remote verification of reported properties; such changes can be incorporated into many existing network devices with little extra cost. Thus, NetQuery can support analysis on a wide variety of networks and cloud data center stacks.

This thesis derives its motivation and design from the shortcomings that manifest in today's federated systems due to their lack of knowledge plane support. We illustrate these shortcomings using both theoretical and real-world examples. We then describe the representation, policy, and analysis infrastructure common to any such knowledge plane.

The work in this thesis carefully preserves existing operator practices and trust relationships. This design constraint stems from a standard assumption in the literature regarding the reluctance of operators to share information. Operators often treat technical details about a network or data center as proprietary because such information can benefit competitors. Though revealing such information may improve coordination and efficiency [105], the potential risks may outweigh such gains. Thus, this poses an economically rational barrier to the deployment of a knowledge plane.

To accommodate existing operator practices and trust relationships, NetQuery limits the information disseminated between domains to sanitized summaries.

Such summaries are expressive, automatically generated, and trustworthy by construction. Sharing only summarized information provides defense in depth: even if knowledge plane functionality fails, due to security breaches or analysis bugs, the consequences are no worse than when an operator today makes an incorrect manual decision based on ad hoc reasoning.

We apply the knowledge plane to solving the identified problems with application analysis; these case studies rely largely on on just those ground properties collected by existing system management infrastructure. These analyses are sound since they rely on ground properties collected by system components that use monitoring to provide high assurance about the accuracy of the properties.

1.3 Supporting reasoning in federated systems

Reasoning is challenging in federated systems because the producers and consumers of knowledge plane information may not trust one another. Indeed, applications that use the knowledge plane to facilitate inter-domain coordination primarily rely on analyses that cross such trust boundaries. These applications make decisions based on the inferred characteristics of systems operated by other administrative domains, typically using information generated from those foreign domains.

For this type of analysis to be meaningful and deployable, a knowledge plane should satisfy two key constraints:

First, all properties and analysis results should be remotely verifiable, thus protecting them from manipulation by an unscrupulous operator. By making claims remotely verifiable, contractual obligations and application requirements can be checked without resorting to manual audit or cumbersome out-of-band mechanisms. The resulting automatic, online procedure reduces economic transaction costs, is precisely defined, and improves assurance while simplifying application design.

Second, the system should respect the information disclosure policies of network operators. Since network operators carefully guard proprietary details about configuration, topology, and traffic matrix, the system should provide mechanisms for mutually untrusting operators to verify one another's claims without leaking confidential data.

1.3.1 Achieving remote verifiability

The central challenge in making properties remotely verifiable is that an unscrupulous operator is a powerful adversary with physical access to the devices under its control. Using this access, such an operator might modify devices, rewire the network, or forge messages so as to falsify some property.

Extending device software and hardware platforms with TPMs and lightweight monitoring mechanisms can prevent such tampering, thereby increasing assurance in the properties reported by devices. An analysis can protect itself from inaccurate information by using only properties that are attributed to trustworthy sources. Typically, attribution is based on TPM *attestations*: attestations are unforgeable certificates binding software-specified statements, such as properties, to an originating hardware/software platform.

A TPM-attested source can be deemed trustworthy by reasoning about the properties of its platform. For instance, a trustworthy platform would protect its control- and data-planes from operator tampering, say by excluding man-in-the-middle attacks and proactively monitoring reported properties for agreement with real-world performance and connectivity observations.

Using TPM attestations has two important benefits. First, it structures analysis to only require trust in device manufacturers. The manufacturers have little incentive to cheat since, unlike the operator, they have little vested interest in the properties advertised from a given deployed network. Establishing trust in manufacturers is also easier, since the effort to establish trust in manufacturer is amortized across all operators using a given device platform. By comparison, establishing trust in operators requires dedicated work to check each individual operator. Second, it allows NetQuery to readily exploit the management information that devices already export by linking each property to a trustworthy platform.

1.3.2 Enforcing information disclosure policies

Providing access to a network's properties to external applications can violate the network's information disclosure policy. Rather than provide direct access to this restricted information, NetQuery mediates access with *sanitizers*, which are analyses that convert restricted information into output properties that are suitable for export. The choice of sanitizer function is straightforward for many applications. For instance, when using NetQuery to check SLA claims, the sanitizer is simply an analysis that examines the network to determine the expected performance, while the output is the computed performance level. When using NetQuery to augment mutual backup agreements, the sanitizer is an analysis that examines each network to determine whether a routing error has occurred. In each case, the information revealed through the sanitizer is equivalent to that revealed by the contract.

To prevent the confidential underlying properties from leaving an operator's control, the analysis is executed on a machine controlled by the operator. Attribution and attestation provide remote verifiability, just as in ground facts from devices. To enable an external application to remotely verify the analysis result, the analysis runs within a *trusted execution environment* that is isolated from the operator and provides an attestation to the analysis binary. As with an externally-run analysis, the sanitizer analysis relies only on properties that it deems trustworthy, thus inductively extending assurances about the trustworthiness of local properties into assurances about the trustworthiness of claims spanning a whole network.

NetQuery can be extended to support analysis that uses information from more than one administrative domain (*multi-domain analysis*) in several ways. One approach is to generalize sanitizers: to compute multi-domain analyses that can be decomposed into independent, per-domain sanitizers, an external client can request sanitizer output from each domain, then use these to derive global system properties.

Another approach that can be used to compute other classes of functions is to combine trustworthy computing with secure multi-party computation (MPC) [156]. Like sanitizers, MPC can perform analysis without revealing confidential information to the participants or to a trusted third party. MPC protocols can potentially improve inter-domain coordination. For instance, using MPC to optimize network configuration across multiple autonomous systems (ASes) can improve network performance in the participating networks [105]. Yet concerns about leaking proprietary provider information poses a barrier to deployment of such optimizations: since a MPC protocol alone is not enough to guarantee all desired security properties, additional application-specific assumptions (e.g., economic arguments) are needed to constrain the attacker [105]. NetQuery is complementary to MPC in that it substitutes these specialized, ad hoc assumptions with standard ones regarding the security of sanitizers and the trustworthiness of properties.

To achieve this substitution of assumptions, all instances of the optimization protocol are executed within trusted execution environments; attestations constrain every participant to running a trusted implementation of the optimization protocol. NetQuery also prevents malicious operators from manipulating the inputs to the optimization: like sanitizers, the trusted implementation accepts only trustworthy information as inputs.

1.4 NetQuery in multi-tenant cloud datacenters

Multi-tenant cloud data centers [15, 110] are emerging as important providers of computational services. Cloud providers deliver network, processing cycles, and storage to customers by hosting customer applications as *tenants* within data centers. Such aggregation of workloads benefits from economies of scale, economies of scope, and statistical multiplexing, allowing cloud providers to profitably deliver scalable, highly available computation at low cost.

Cloud abstractions such as virtualization enable application code and data to readily migrate between different locations and providers. Compared to traditional applications, whose choice of network providers is limited to those that serve their fixed locations, cloud-hosted applications have more agility in choosing cloud providers. Providing system services, such as a trustworthy knowledge plane, for disseminating information about different providers can help prevent decisions about provider suitability from limiting agility.

Because network services are a central component in delivering the cloud service model, the ability to reason about network properties can provide benefits to cloud-hosted applications. For cloud-hosted applications that deliver services over the Internet, the quality of WAN access directly impacts end-to-end performance; NetQuery presented thus far can support such reasoning. Yet in a data center, the performance and availability of applications depends on the combination of network, processing, and storage components. For example, large tenant applications often depend on scalable, multi-tier system architectures, which heavily stress each of these components. To enable cloud-hosted applications to reason about all of their infrastructure dependencies, a knowledge plane for the cloud should disseminate properties about all such components.

NetQuery is well-suited to the architecture of typical virtualized cloud datacenters. Such datacenters are logically split into pools of resources, such as network, computational, and storage, along with a fabric controller [98, 114, 112] that manages these resources. Computation and storage nodes are implemented on commodity server hardware, for which TPM-bearing motherboards are a standard option. They are also managed by the fabric through a narrow interface (e.g., VM creation and deletion, virtual switch reconfiguration), with well-defined, standard schemas for describing their state (e.g., list of all active VMs and active firewall rules). Thus, the computation and storage properties of a datacenter can be exported as attestation-backed, attributed properties.

1.4.1 Datacenter analyses

Analysis over these datacenter schemas can be used to verify that a datacenter satisfies many types of claims, such as those governing customer-to-datacenter and datacenter-to-datacenter interactions. As in federated networks, many claims that are challenging to measure from the tenant's vantage point are straightforward to establish with analysis.

Datacenter tenants can use analysis to differentiate between multiple cloud providers based on performance metrics. For instance, a tenant can analyze the network topology and number of active VMs to determine maximum network capacity, oversubscription level, and redundancy, the latter two of which require information that cannot readily be found with probing. Using these results, a tenant can pick the most appropriate provider for a given application.

Tenants can also use NetQuery to verify contractual obligations. For instance, some industry security standards mandate segregation between nodes and networks that handle data with different classification levels [9], while regulations can mandate that data be maintained within a specific legal jurisdiction. Some cloud providers sell premium units of computation that are indistinguishable in function and performance from regular computation yet have significantly different isolation and provisioning properties. For instance, tenants may pay extra for VMs that are physically segregated from other tenants, minimizing potential for information leakage [12]. Tenants may also pay to reserve physical hardware for future use [14]. In both cases, analysis over hypervisor properties can verify that these promises are kept.

Trading computational capacity on commodities exchanges can improve pricing efficiency and streamline discovery and exploitation of idle resources [153]. NetQuery provides several important mechanisms for supporting such trades. Commodities are typically categorized into equivalent quality bins to abstract away unimportant details and to maximize fungibility and liquidity; the performance metrics, such as those described earlier, can do this. NetQuery also provides accountability to exchange-mediated transactions, reducing the likelihood of fraud.

1.4.2 Extending the data center data plane with new guarantees

Performance predictability is an important concern when migrating applications to cloud. While multiplexing a datacenter across many applications decreases cost and increases provisioning flexibility, managing the resulting performance interactions becomes challenging. TCP is the dominant mechanism used to apportion network resources; TCP operates on a flow granularity, allocating network capacity in proportion to the number of TCP flows opened by a given application. However, a TCP-based allocation policy is prone to abuse: by simply opening many TCP flows, selfish or malicious applications can hoard network capacity or launch denial of service attacks. Thus, deriving claims about performance is challenging given these inadequacies.

This thesis proposes several new protocols for allocating resources in cloud datacenters. These protocols run within *trusted packet processors*, which are programmable packet processing elements that are isolated from applications and operators and support attestation. The network infrastructure uses these processing elements to augment the fixed function hardware forwarding path with new functionality, such as the resource allocator algorithm. Trusted packet processors provide assurances to the infrastructure and to external applications. Strong isolation from malicious tenant code gives the network infrastructure the assurance that new functionality is executing properly. With appropriate attestations and configuration analysis, a data center can assure tenants and external NetQuery applications that new functionality is in place. For instance, by checking for the presence of a resource allocator with a particular set of parameters, an application can verify performance claims.

This thesis examines two types of trusted packet processors that are deployable in today's datacenters with incremental software changes and little to no change to switch hardware.

- **End host virtualization stacks.** Trusted execution environments at the end host can readily inspect all traffic for a given node; when run on the host CPU, they benefit from the flexibility of a general purpose computing platform. For instance, filter modules attached to the virtual switch can monitor and interpose on all traffic entering and leaving VMs and can be implemented with few changes to the virtualization stack.

  Using filter modules, we have implemented Seawall, a network bandwidth allocator for cloud datacenters. Seawall allocates bandwidth using a distributed congestion control algorithm that avoids the problems of TCP by using a link-oriented, rather than a classic flow-oriented, congestion control scheme. Seawall supports policies for specifying relative bandwidth allocations between different VMs; these policies can be used to isolate workloads, provide differentiated service models, and optimize network scheduling in compute clusters.

- **In-network sidecar processors.** Whereas end host virtualization stacks provide new control mechanisms and vantage points at the edge, trusted execution environments located at switches provide new control mechanisms and vantage points in the middle of the network, allowing direct observation of events which might otherwise be invisible or need to be inferred. This thesis proposes SideCar, a programming model that enables programs to deploy custom processing code at any switch in the datacenter. SideCar programs can specify special processing for a fraction (1-10%) of traffic at any switch. The processing code is an arbitrary program that executes within a sidecar control processor attached to each switch. SideCar programs are specified as set of predicate/action rules: the predicate specifies a pattern match for packets traversing the switch and the action rule specifies a monitoring or rewriting action to take on a matched packet.

  SideCar's constrained programming model can be retrofitted to commodity switches with minimal cost increase, little disruption to regular forwarding, and no special modifications to the control software. By comparison, systems that support a fully-programmable, general purpose network forwarding path for all traffic [53, 81] require significantly costlier hardware than the relatively fixed-function datapaths of switch ASICs [104]. Though SideCar is limited to processing a fraction of the network capacity, it supports many applications, including providing more precise feedback for

Seawall's control loop and enforcing IP/Ethernet reachability isolation and SAN bandwidth allocation. Such applications expand the set of security and performance claims that a cloud datacenter can export to its tenants.

1.5 Summary

This thesis describes the design and implementation of a knowledge plane suitable for disseminating and reasoning about federated systems, such as the Internet and cloud computing infrastructure. Knowledge plane-enabled systems can export remotely verifiable claims about their behavior to external applications, customers, and peers. Existing techniques for making claims about system behavior rely heavily on manual, ad hoc checks; the knowledge plane replaces these with automatic verification based on analysis of the system in question. Such analysis can verify system properties that are difficult or infeasible to verify with probing-based techniques.

Overall, this thesis shows how remotely verifiable claims are a powerful tool for inter-domain coordination and describes how this abstraction can address standing problems in enterprise networks, the Internet, and cloud services. We have built NetQuery, a knowledge plane for the Internet, along with several devices and applications. We have deployed NetQuery on a real departmental network and evaluated it against real-world Internet traces and topologies. We also present network mechanisms that provide novel guarantees over network resource allocation. We have built Seawall, a bandwidth allocator that provides cloud operators and applications with flexible control over apportioning of network capacity, and evaluated Seawall on a representative data center testbed.

1.6 Thesis outline

This thesis is structured as follows. Chapter 2 presents the motivation and design of NetQuery and covers the data model and logic for generating and reasoning about properties, the trustworthy computing techniques used to achieve remote verifiability, and mechanisms for controlling information disclosure. Chapter 3 describes how NetQuery uses hardware and software stacks for trustworthy computing to implement knowledge plane abstractions; the chapter also describes the security model and its design implications. It also describes the applications that NetQuery enables in federated networks such as the Internet. Chapter 4 presents an evaluation of the NetQuery prototype. Chapter 5 describes how to extend NetQuery to virtualized cloud datacenters and outlines the benefits. It then presents the system architecture and protocols underlying Seawall and SideCar. Chapter 6 presents the Seawall algorithms in detail along with an evaluation of the prototype. Chapter 7 places NetQuery in the context of related work and Chapter 8 concludes.

CHAPTER 2

NETQUERY: A KNOWLEDGE PLANE FOR FEDERATED SYSTEMS

Knowledge planes enable new applications that depend on reasoning about properties of the network. By providing mechanisms for determining the characteristics of a network, such as the expected level of performance, redundancy, or confidentiality, NetQuery enables network participants (e.g., peers, providers, or customers) to make better informed decisions when establishing sessions, entering into contracts with one another, or verifying compliance.

2.1 Motivation and overview

Sound network reasoning improves network transparency and accountability, facilitating many types of commercial transactions. On the Internet, the price that an operator can charge depends on the paths and performance that they advertise; since routing traffic on a different path may result in lower cost, operators are incentivized to deviate from their advertised behavior to maximize profit [68]. Existing reputation-based mechanisms have not proven sufficient to constrain selfish operators. Indeed, consumer advocates often accuse last mile ISPs of misrepresenting the quality and capacity of their networks [30]. Community forums often advise users to verify independently whether their datacenter provider is truly multi-homed [48]. Differences between networks are advertised through manual, ad hoc channels such as interstitial web pages. Absent automatic mechanisms for discovering and disseminating claims about network capacity and redundancy, agreements are difficult and costly to enact and competitors can engage in unscrupulous practices. With NetQuery, ISPs

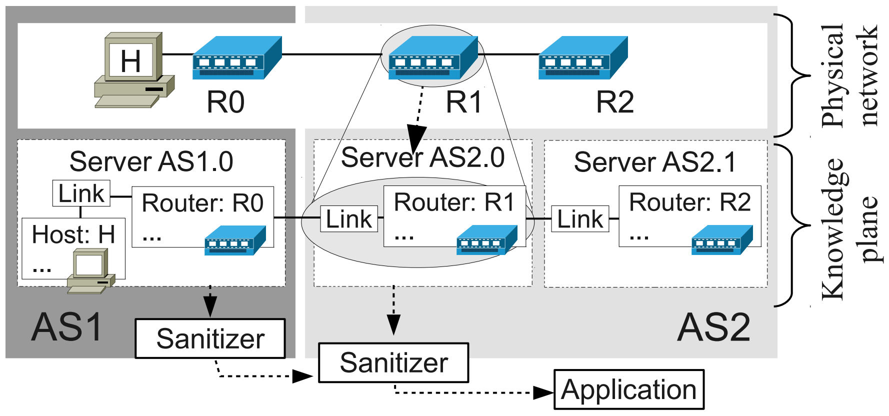


Figure 2.1: **NetQuery architecture.** A physical network and its knowledge plane representation, stored on knowledge plane servers operated by each AS.

with good networks can advertise the quality of their networks in an automatic, remotely verifiable fashion.

Transparency and accountability for claims improve market efficiency by reducing the economic transaction costs associated with establishing agreements. In particular, NetQuery allows applications to discover network properties that are otherwise difficult or impossible to determine by external data plane probing.

NetQuery enables applications to reason about properties that span multiple ASes. Since NetQuery's logic-based credential system supports many mechanisms for establishing trust besides a priori relationships, such reasoning can readily incorporate information from any principal. For instance, TPM-based credentials leverage trusted hardware to incorporate device-generated information, and audit-based credentials incorporate network information added by trusted third parties.

2.1.1 System overview

Network information is disseminated in NetQuery using a *knowledge plane* that maintains a representation of the network topology and configuration (Figure 2.1). The knowledge plane makes this information available to applications for determining whether the network exhibits desired properties. The process of inferring some high-level *characteristic* of the network (such as the loss rate on a route between hosts) from low-level properties (such as routing tables) is called *analysis*. Status information about network entities is typically self-reported by these entities (e.g., routers export their forwarding tables), though a transition mode is supported for proxies to transfer management information from legacy entities to NetQuery.

Logically, a single, global knowledge plane incorporates all properties across multiple administrative domains on the Internet. Physically, this knowledge plane is federated — each administrative domain runs a cluster of servers that locally stores all information describing its network. Federation facilitates incremental deployment and protects confidentiality, since an operator can independently run NetQuery servers without making information accessible to operators of other administrative domains.

Applications can query the knowledge plane for information about any participating network. Networks will typically restrict direct access from external parties, but instead allow *sanitizers* to execute operator-authorized sets of analyses on behalf of external applications. These analyses export only the network characteristics that meet an operator's information disclosure policies. The exported sanitizer results are accompanied by credentials certifying that the correct analysis code was executed.

Each NetQuery application independently defines the set of principals it trusts. The knowledge plane can include conflicting information from different sources; applications can filter such information based on the principals that they trust. This trust can be predicated on any credential associated with a principal. Often, such credentials are issued by a TPM, which binds statements issued by a principal to a hardware/software platform. TPMs enable NetQuery to collect a broad pool of attributed properties at low cost. The per-unit hardware cost of TPMs is low, which permits wide deployment. With modest computational cost, TPMs can automatically publish information to the knowledge plane at short time intervals, with each statement bound in an unforgeable certificate; by comparison, manual audit is costly and infrequent.

Attestation also enables trust establishment costs to be amortized. In particular, rather than establish trust with every counterparty, analyses need only establish trust with platforms, with this cost amortized across all users of those platforms. Engineering constraints help to limit the total number of software and hardware platforms in use: reusing hardware and software components reduces development costs for vendors, and deploying a small set of standard platforms reduces testing, support, and maintenance costs for vendors and users.

Often, there are multiple ways to satisfy a desired network characteristic. For instance, the fault resilience of a network may be derived through an analysis of its topology or through independent vetting by an auditor. NetQuery employs a logic for sound, flexible derivation of characteristics. Logical reasoning yields a proof that is self-documenting in that it describes all the assumptions and inference steps used to conclude that a characteristic holds. Such proofs are useful in logging and auditing.

Overall, NetQuery incorporates several key abstractions and implementation techniques that facilitate its deployment on existing federated networks. Sanitizers simultaneously protect confidential operator information yet remain expressive enough to discharge many existing contractual claims and stipulations. The low-level system information that is already collected and exchanged in existing networks for monitoring and management provides a rich set of properties with which to drive analysis. Enhancing such mechanisms with new trustworthy computing abstractions, backed by TPMs, enables the knowledge plane to accumulate a comprehensive view of the network at low cost and high assurance.

2.2 Data model

A key challenge in building a knowledge plane for federated systems is to handle multiple trust domains. A knowledge plane that spans multiple organizations must support access control policies to protect confidential properties. Because the knowledge plane might contain inaccurate information, applications should be able to express policies that specify what information is safe to use for analysis. Since the system includes a diverse range of devices and implementations, standardization of nomenclature is necessary for interoperability.

The knowledge plane in our NetQuery prototype is based on a *tuplespace* representation. Every *tuple* is named by a globally unique *tuple ID* (TID) and stores properties as typed attribute/value pairs; an attribute/value pair and its associated metadata is called a *factoid*. NetQuery supports string, integer, references to tuples, dictionary, and vector values for factoids.[1]

[1]A production version of NetQuery might well leverage the ongoing development of semantic web technologies such as RDF and OWL [116] for building the knowledge plane.

NetQuery *principals* are the basis for policy decisions. Every producer and consumer of information from NetQuery is represented by a principal, which has a unique public/private key pair. The key pair is generated independently. NetQuery records two pieces of policy-associated metadata for every factoid it stores: *attribution*, which captures the principal responsible for generating the factoid; and *export policy*, which defines what principals can read the factoid.

In a federated environment, it is impractical to expect global consistency or uniform interpretation of properties contributed by diverse sources. But by retrieving attributions, applications have some basis to reason about whether a property is suitable for use based on whether the provider of that property is trustworthy.

To facilitate interoperability between devices and analyses, NetQuery information conforms to voluntary *schemas*. Each schema defines the set of properties that a given kind of network element must provide. NetQuery schemas prescribe a data format but do not prescribe associated code or operations. NetQuery provides standard schemas for representing devices (e.g., hosts, routers, and switches) and network abstractions (e.g., TCP connections, TCP/UDP endpoints, and IPsec security associations). NetQuery schemas are similar to those of network management systems (e.g., SNMP) that are supported by many devices. By adding to such devices a shim for outputting properties or by interfacing through an SNMP proxy, we enable them to participate in NetQuery.

Research into a semantic web has produced considerable infrastructure and theory for the federated knowledge stores, reasoning, and query processing that underpin any knowledge plane. NetQuery can also help ongoing efforts to extend the semantic web to cover network management information.

Initializing factoids Tuples and factoids for a network device can be initialized and maintained by different network participants, including the device itself, its administrator, or a third party. Since routers and switches have limited processing capacity, they offload their tuples to *tuplespace servers*, thereby insulating against application-induced query load.

On start up, a device discovers its local tuplespace server from DHCP and transfers local configuration and initial state to the tuplespace by issuing **create**() operations to the discovered tuplespace server to instantiate tuples and **update**() operations to write factoids. A newly activated router, for example, creates tuples for all of its interfaces and exports its initial forwarding table state. Hosts, routers, and switches also export local topology information to the tuplespace, using a neighbor-discovery protocol such as LLDP to generate ground facts. The device pushes any changes to its configuration and state to the tuplespace server.

Information sources can also generate tuples and factoids on demand by registering a callback function with a tuplespace server. This mode of operation is well-suited for exporting large sets of properties, such as forwarding tables, that are costly to precompute and export en masse to the knowledge plane.

Tuplespace servers and lookup protocol Each AS or third-party information service operates tuplespace servers. The TID for a tuple embeds the IP of the tuplespace server storing that tuple, along with an opaque identifier. Device references are stored as TIDs, and therefore analysis can efficiently access the relevant tuplespace servers. To prevent changes to facts and metadata while in flight, tuplespace servers communicate over secure channels.

To prevent DoS attacks by applications that issue costly tuplespace server operations, all remote operations always terminate in bounded time. The tuplespace server only supports simple wildcard queries for attributes, which in the worst case loops over all attributes for a given tuple. NetQuery provides no mechanisms for invoking either recursive queries or stored procedures. Clients, however, are free to perform processing locally, including data aggregation and other expensive analyses.

Tuplespace servers make extensive use of soft-state to improve performance. Since the tuplespace contents derive from device state, a tuplespace server can always ask devices to re-export their state. This obviates the need for tuplespace servers to support a costly transactional recovery mechanism. The tuplespace uses lease-based storage management for factoids. Thus, once a device fails, the stale factoids will be garbage collected automatically.

2.2.1 Dynamics: changes and triggers

Changes in the described federated system can invalidate properties in the knowledge plane. Time of check/time of use bugs can occur because system changes take place concurrently with analysis. Similar behavior can arise from probe-based network measurements such as `traceroute`.

NetQuery provides a *trigger* interface to facilitate detection of changes to underlying properties during analysis. In addition to this, the interface allows applications that depend on long-running characteristics to be notified when some conclusion no longer holds. Once a trigger is installed, a callback packet is sent to an application-specified IP-port pair (*trigger port*) whenever a specified factoid has been modified. Triggers are stored by tuplespace servers as soft-state, so applications must periodically send keep-alives to refresh their triggers.

Our NetQuery applications check for relatively stable characteristics; here, triggers suffice to filter spurious analysis results that arise from observing transient states. Tools built using probing interfaces typically would be fooled by such transients. To implement this filtering, applications install triggers for factoids being used. For instance, consider an application that enumerates all hosts in some given L2 Ethernet domain by issuing queries that traverse the network graph. Were the topology to change during these queries, the application might miss newly connected hosts. To guard against this, the application issues a **retrieve_and_ install_ trigger**() operation to atomically retrieve link information from a switch and thereafter to monitor it for changes. This atomic operation eliminates the window of vulnerability between the time a factoid is read to when monitoring for changes to that factoid starts.

Network delays in message delivery can cause updates from different devices to be received at a tuplespace server interleaved in unpredictable ways; eliminating such inconsistent views would simplify analysis. For systems that have a built-in notion of state consistency, such as distributed network control planes [97] and cloud fabric controllers [114, 98], knowledge plane consistency can derive from this existing state consistency, for instance, by adding a consistent cut guarantee. Other systems, such as the Internet, are built from protocols and devices that support neither consistency nor causality. We could augment devices with logical clocks, but that would be challenging to deploy incrementally, since means would be needed to approximate causal dependencies when exchanging messages with legacy devices. Fortunately, operator reasoning in the network domain is typically focused on steady-state network characteristics, and causal consistency is less critical there.

2.3 Analysis

NetQuery provides mechanisms that parse factoids acquired from tuplespace servers, determine whether a factoid is trustworthy, and check if a desired network characteristic is supported by trusted factoids.

Nexus Authorization Logic (NAL) provides the logical foundation for making inferences from factoids. A full discussion of NAL is beyond the scope of this dissertation (see [133]). Below, we simply outline the main features of NAL, describe how it is used in NetQuery, and discuss the implications of our choice. NAL admits reasoning about factoids and attribution information, enabling applications to reconcile conflicting statements uttered by different principals. The says and speaksfor operators, along with a set of two dozen inference rules, permit inferences based on an application's trust assumptions.

NAL associates a *worldview* with each principal. This worldview contains the beliefs a principal holds about the network. Reasoning in NAL is local to a worldview: by default, inference takes into account only local beliefs, rather than statements believed by other principals. Local reasoning prevents reasoning by one principal from being corrupted by contradictory facts attributed to another principal. Using speaksfor, a principal can import into its worldview beliefs from other principals that it trusts. An optional scoping parameter restricts speaksfor to import only statements concerning a specific matter of interest.[2]

[2]Since NAL does not encode a notion of degrees of trust, speaksfor is monolithic in that

Applications use NAL to derive theorems about the underlying federated system from information provided by the NetQuery knowledge plane. Specifically, factoids are converted to logical statements, which are then used to prove some given *goal statement*. A goal statement is a NAL formula that characterizes what a client application wants to establish. An application initially populates its worldview with an *import policy*, specifying what factoids the client deems to be trustworthy. Import policies often include speaksfor statements that specify which hardware and software platforms are trusted sources for factoids.

Ground statements Factoids from tuplespace servers translate into NAL *ground statements*. Tuplespace API operations, such as fetching factoids, translate their return values to NAL formulas. For instance, a **retrieve**() operation issued on router *R0*'s tuple returns factoids as NAL axioms of the form:

$$\begin{aligned}&\mathit{TuplespaceServer}\ \textsf{says}\\ &\qquad\qquad \mathit{R0}\ \textsf{says}\ (\mathit{R0}.\texttt{Type} = \text{“Router”} \land\\ &\qquad\qquad\qquad \mathit{R0}.\texttt{fwd\_table} = \{\mathit{ip}_A \Rightarrow \mathit{nexthop}, \ldots\})\end{aligned}$$

where "ip_A => $\mathit{nexthop}$" denotes a forwarding table entry specifying the next hop for a given destination. The nesting of says formulas captures the full chain of custody during the factoid's traversal through the knowledge plane. Here, the factoid was exported from *R0* to a particular tuplespace server, *TuplespaceServer*.

The NAL *proof* for a goal statement is a derivation tree with the goal statement as the root, NAL inference rule applications as internal nodes, and ground statements and axioms as leaves. Analyses typically provide a proof generator

it incorporates all in-scope statements. NetQuery can switch to a logic that supports such reasoning [40, 107] should the need arise.

that embodies a programmer's understanding of how to check whether a given characteristic holds into a proof generation strategy.

Proofs can be consumed entirely within a single application or exported to other parties as a self-documenting certificate. The certificate can be logged to create an audit trail for accounting, documentation, and debugging. Such audit trails are also useful in application domains governed by external compliance requirements, such as Sarbanes-Oxley and HIPAA.

2.3.1 Generating proofs

Since general-purpose theorem provers are typically infeasible, analyses normally embed a custom proof generator for a specific set of claims. Such proof generators are often straightforward to adapt from existing code by deriving the proof generation strategy from the structure of existing analyses and control plane code.

Proof generators can be built using several design patterns. One such design pattern is to certify the execution of a program on specific inputs, using attestation in a manner similar to [135]. This program is trusted to check the claim in question and can be written in an arbitrary programming language. For instance, an industry-standard script for finding misconfiguration errors [61] can be adapted using this design pattern to prove that a system is properly configured.

Whereas the preceding design pattern wraps an external checker within a few NAL statements, another design pattern is to fully describe the verification process with NAL statements. Where the characteristics being inferred closely match the correctness assertions of existing system functionality, such assertions could be translated to NAL derivation steps. For instance, consider the operation of starting VM on a given node. The pre-condition of this operation is to verify that the target node has sufficient CPU and memory; the post-condition is that the VM is loaded and running on the node. Both of these checks can be converted to a NAL proof showing that the VM was provisioned with sufficient resources.

2.3.2 Example: Checking network paths

This example shows how an application might use NAL to verify that a network complies with a performance requirement. Suppose site *A* wants to establish that the path to site *B* provides low loss rate (Figure 2.2).

The goal statement for a path *P* to satisfy a bound *r* on the loss rate is

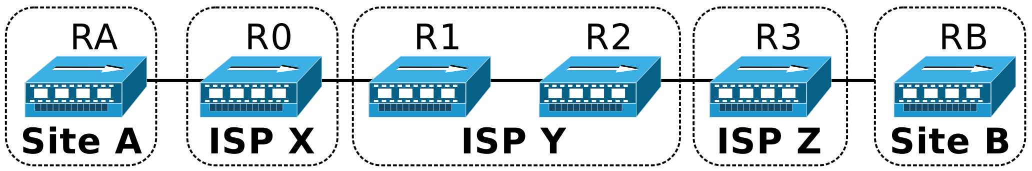
$$\begin{aligned}
&\exists P \in \mathit{Routers}^{2+} : \\
&(P[1] = A) \ \land \ (P[|P|] = B) \ \land \\
&(\forall i \ 1 \leq i < |P| : P[i] \rightarrow \texttt{fwd\_table}\langle B \rangle = P[i+1]) \land \\
&(\exists r' \in \mathit{Reals} : (r' < r) \ \land \ \text{TrustedLossRate}(P, r'))
\end{aligned} \tag{Goal}$$

We write "$\mathit{Routers}^{k+}$" for the set of all finite router-tuple sequences of length *k* or greater, "`TID.name1` $\rightarrow$ `name2` $= v$" as shorthand for dereferencing a reference value factoid to access a second factoid, "$|P|$" for the length of a sequence, "$P[k]$" for the *k*th element in a sequence, and "$T\langle key \rangle$" for a lookup from a factoid of dictionary type. The first three lines constrain *P* to be a valid path given the source, destination, and forwarding table state. The last line asserts an upper bound on the expected one-way loss rate on *P* given information from trusted principals, and that predicate is defined as

$$
\begin{aligned}
&\text{TrustedLossRate}(P, r) \triangleq \\
&\quad (\exists P', P'' \in \mathit{Routers}^{1+}\ \exists p \in \mathit{Routers}\ \exists r', r'' \in \mathit{Reals} : \\
&\qquad (P = \text{Concat}(P', p, P''))\ \wedge\ (r = r' + r'')\ \wedge \qquad \text{(Cjoin)} \\
&\qquad \text{TrustedLossRate}(\text{Concat}(P', p), r')\ \wedge \\
&\qquad \text{TrustedLossRate}(\text{Concat}(p, P''), r''))\ \ \vee \\
&\quad (\exists R, R_{next} \in \mathit{Routers} : \\
&\qquad (P = \text{Concat}(R, R_{next}))\ \wedge \qquad \text{(C0)} \\
&\qquad (R\ \textsf{says}\ R.\texttt{curr\_loss\_rate\_to}\langle R_{next}\rangle = r)\ \wedge \\
&\qquad \text{IsTrustedRouter}(R))\ \ \vee \\
&\quad (\exists I \in \mathit{ISPs} : \\
&\qquad (\forall i\ 1 \le i \le |P| : P[i].\texttt{ISP} = I)\ \wedge \qquad \text{(C1)} \\
&\qquad (I\ \textsf{says}\ I.\texttt{sla\_loss\_rate}\langle P[1], P[|P|]\rangle = r)\ \wedge \\
&\qquad \text{IsTrustedISP}(I))\ \ \vee \\
&(\exists A \in \mathit{Auditors} : \\
&\quad (A\ \textsf{says}\ A.\texttt{measured\_loss\_rate}\langle P[1], P[|P|]\rangle = r)\ \wedge \qquad \text{(C2)} \\
&\quad \text{IsTrustedAuditor}(A))
\end{aligned}
$$

where "Concat()" denotes sequence concatenation. IsTrustedRouter(), IsTrustedISP(), and IsTrustedAuditor() are predicates derived from import policies, where the policy for IsTrustedRouter() checks the attestation and platform while the latter two check whether I and A are on a whitelist of trusted principals.

This analysis incorporates facts from multiple trustworthy sources, each having different ways to determine the loss rate of a path: (C0) expresses trust in certain routers to report their instantaneous link statistics, (C1) expresses trust in certain ISPs to claim SLAs, and (C2) expresses trust in certain auditors and measurement tools [106] to report measured performance. Each rule infers loss rate using different data schemas.



< RA.*fwd_table* = { RB => R0, ... } >
< R0.*curr_loss_rate_to<R1>* = 0.1%,
  R0.*fwd_table* = { RB => R1, ... } >
< R1.*ISP* = ISP-Y,
  R1.*fwd_table* = { RB => R2, ... } >
< ISP-Y.*sla_loss_rate<R1,R3>* = 0.1% >
< R2.*ISP* = ISP-Y,
  R2.*fwd_table* = { RB => R3, ... } >
< R3.*ISP* = ISP-Z,
  R3.*fwd_table* = { RB => B, ... } >
< A.*measured_loss_rate<R3,RB>* = 0.1%>

Figure 2.2: **Topology and tuplespace contents for network analyzer example.** Attribution metadata ("says" information) has been omitted for brevity.

2.3.3 Using TPM attestations

Every TPM is uniquely identified by a set of keys that is certified by a PKI being operated by the device manufacturer. Attestations are remotely verifiable, unforgeable, and tamper-proof certificates that use these device keys to bind a software-generated bit string (typically, a message) to the particular hardware and software platform that generated it [66]. By linking attestations to attribution metadata, NetQuery allows applications to unequivocally link each factoid back to some particular device.

Attestation is not a panacea against all misbehavior. Attestation merely establishes accountability; a NetQuery client using a factoid has to decide whether to trust the platform that is attesting to that factoid. For instance, suppose a routing platform is designed to honestly report its observations of the control- and data-plane as factoids. It is tempting to assume that attestation of this platform implies that the factoids agree with the real-world. But malicious operators can manipulate these observations and trick poorly-constructed devices into issuing factoids that disagree. Section 3.2 discusses such attacks.

TPM optimizations

NetQuery applications make their own determination about how factoids are interpreted based on attribution information. In the baseline system, every factoid and credential is signed, allowing clients to independently check that factoids are attributed to the right principals and that credentials are issued by the right parties. However, this checking is costly if clients use factoids and credentials from many different principals. Hence, NetQuery incorporates two optimizations.

Avoid TPM signatures Whenever possible, NetQuery avoids obtaining signatures from the TPM, which is much slower than the CPU. Instead, NetQuery constructs software principal keys using a TPM-rooted certificate chain that the CPU uses for signing every factoid update. Thus, TPM attestations are used only once per boot, to bind the software principal to the platform.

Attest to tuplespace servers Signatures provide end-to-end integrity and authentication for factoids. This helps when tuplespace servers might manipulate factoids, but it is unnecessary for tuplespace servers that are trusted to relay factoids correctly. By distinguishing such servers with attestation, NetQuery can replace per-factoid signatures with secure channels built from symmetric keys. Clients using this optimization specify an import policy that accepts tuples without signature-verification from attested tuplespace servers. These policies leverage the `says` information within ground statements, which encodes the tuplespace server's position as a repository of utterances from other principals.

2.3.4 Confidentiality and sanitizers

Agreements between participants in a federated system often stipulate the presence of certain network features. For instance, network peering agreements mandate up-time and fault tolerance guarantees [18], SLAs mandate desired latency and loss rate characteristics, and service agreements for cloud datacenters reference the network bisection bandwidth and oversubscription levels. Verifying the accuracy of such advertised claims is at best difficult and often impossible. Trust establishment is typically performed manually, pairwise for each agreement, using ad hoc means.

In contrast, knowledge plane analysis can verify such claims in a principled fashion. However, many ASes and cloud operators have strict disclosure policies about internal network information. A naïve use of NetQuery, where external parties run analysis to verify properties of interest, can reveal detailed internal information. To be practical, a knowledge plane needs some way to provide assurances to external parties without revealing confidential data.

NetQuery provides *sanitizers* for this purpose. A sanitizer is a service that converts factoids covering proprietary information into unforgeable summary factoids suitable for release to other parties (Figure 2.3). Sanitizers execute analysis code on behalf of the remote party. To provide assurance that a remote analysis is performed correctly, each sanitizer executes in a trusted execution environment and provides an attestation certificate that ties the output factoid to the sanitizer binary and the execution time. This approach does require both parties to agree on the sanitizer at the time of contract establishment, but once that agreement is in place, a sanitizer that checks the contractual stipulations reveals no more information than what is revealed in the contract itself.

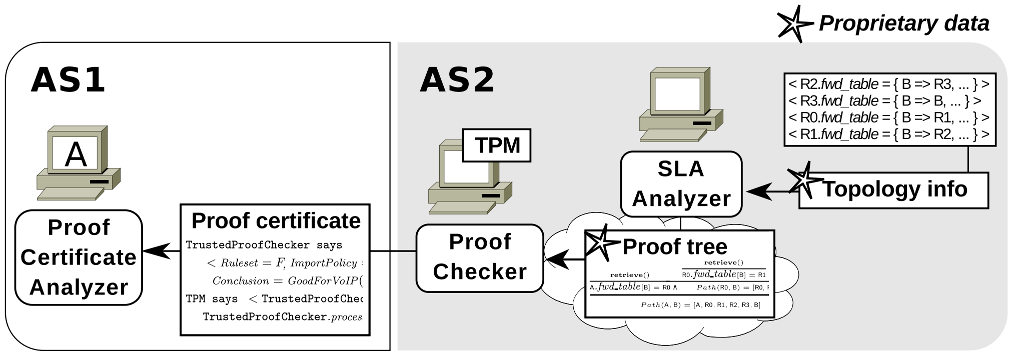

Proprietary data
AS1
AS2
A
Proof Certificate Analyzer
Proof certificate
TrustedProofChecker says
TPM says
TPM
Proof Checker
SLA Analyzer
Proof tree
Topology info
< R2.fwd_table = { B => R3, ... } >
< R3.fwd_table = { B => B, ... } >
< R0.fwd_table = { B => R1, ... } >
< R1.fwd_table = { B => R2, ... } >

Figure 2.3: **Preserving confidentiality.** Sanitizers support external clients without leaking information, since the exported certificates serve as trusted proxies for the full proof.

NetQuery provides confidentiality guarantees through its careful use of the trusted execution platform. A sanitizer executes on an AS's own computers, so factoids that it processes never leave the AS's custody. The trusted execution environment is used solely to provide an execution-integrity guarantee to a remote party. The data confidentiality guarantees then come from the sanitization embedded in the analysis itself—not from the underlying operating system. Using sanitizers means that NetQuery does *not* require an execution environment that can provide confidentiality guarantees against attackers with physical access to the execution hardware—such systems are difficult to build, even with TPMs. Further, since ASes have complete control over when, where and how often sanitizers execute and how much data they reveal to external parties, an AS can prevent outside applications from crafting query streams that consume excessive system resources or that induce a sanitizer to leak information.

While the sanitizer abstraction introduced earlier is very flexible, it requires a priori agreements on mutually trusted, monolithic sanitizers. This approach can be brittle since it commits the local and remote parties to specific sanitizer implementations; it also inflates TCB size, since monolithic analyzers can be large. NetQuery further simplifies the process of securely performing analysis across trust boundaries by partitioning the sanitizer into a *proof generator* and a *proof checker* that both run in the provider (Figure 2.3). The proof generator constructs NAL derivations that are then verified by the proof checker. This partitioning allows a remote party to only have to trust a proof checker—the proof generators need not be trusted, and the local party (i.e., operator using the sanitizer to support a claim) can use whatever sound means to prove that their network possesses a critical property. TCB size is now reduced significantly, since one well-audited proof checker suffices for all NAL analyses, while a custom proof generator is chosen by operators as needed for specific tasks.

Since proof generators appear between proof checkers and factoid sources, a proof generator can potentially inject stale or forged ground statements. To protect against such attacks, ground statements include a nonce, uniquely generated by the client for each proof, and MAC, which are returned with factoids on **retrieve**(). Proof checkers exchange private MAC keys with the tuplespace server and pass nonces for every new proof to proof generators. The checker accepts ground statements, and thus accepts the proof, only if all embedded nonces and MACs are current.

2.3.5 Confidentiality-preserving applications

The following examples show how NetQuery supports different applications while preserving the confidentiality of factoids. These applications, along with the ones described in Section 3.5.1, can all be implemented with sanitizers.

Verifying performance and reliability guarantees Configuration generators that use global network optimization to achieve performance and reliability goals are increasingly prevalent [134, 19]. These tools automatically configure a network based on workload, topology, and performance constraints. Operators that rely on configuration generators can use NetQuery to advertise the achieved goals.

Configuration generators are typically complex and proprietary to an operator; so they are not disclosable to network clients as a means of certifying performance goals. But a sanitizer could run an industry standard configuration checker (such as [44]) to verify that the output configuration from a configuration generator meets the performance claim. Moreover, a network operator can upgrade to a new configuration generator without updating the contract or disclosing the new code.

As an example of this construction, suppose an operator offering MPLS-based VPNs advertises guaranteed bandwidth in the presence of a single node or link failure [88]. The operator could run a global optimizer to find an assignment of MPLS primary and backup paths that satisfies all reservations and link capacity constraints. A configuration checker can then validate this assignment by walking through the MPLS-related factoids: for each failure scenario, the checker would verify that backup paths do not overload any links.

Dynamic verification of contractual obligations Some contractual obligations are easier to verify dynamically than statically. For instance, precomputed backup paths in MPLS VPNs provide bandwidth guarantees only for the first failure. To establish resilience against additional failures, the operator needs to compute a new set of primary and backup paths after each failure. A configuration checker can detect this by using triggers and updating its output factoids accordingly.

But updating the output factoid every time the provider-topology changes leaks information about outage and maintenance intervals. To prevent this disclosure, the operator can interpose another sanitizer certifying that the network successfully recovers from failures within a reasonable time. As long as this assertion is met, the sanitizer leaks no information beyond that found in the customer contract.

CHAPTER 3

USING HARDWARE AND SOFTWARE FOR TRUSTWORTHY COMPUTING

Operators are not only well-positioned to launch attacks on NetQuery but they have incentives to do so. In this chapter, we outline the NetQuery security model and discuss the applicable results from trustworthy computing research in building routers that can provide assurances. We also discuss the vulnerabilities that can exist in analyses that improperly interpret knowledge plane information, along with defenses to protect against such concerns.

3.1 Security model assumptions

NetQuery depends on the following security assumptions:

- **Hardware and software security.** Attackers cannot tamper with the execution semantics of a device. The only way to affect running code is to use the explicit interfaces provided by the device (e.g., I/O ports, RPCs, configuration console) rather than side channels (e.g., installing hardware probes on its memory bus). Moreover, attackers cannot extract secrets stored in secure coprocessors.
- **Cryptographic algorithm security.** This assumption, shared by most work on security, implies that digital signatures used for attestation cannot be forged. It also implies the confidentiality and integrity of messages conveyed by secure channels.
- **PKI security.** An attacker cannot cause the PKI that issues TPM certificates to issue arbitrary credentials nor can it steal the PKI's root keys.

Together, these assumptions imply that TPM attestations are unforgeable, since execution, encryption, and credentials cannot be compromised or spoofed. Consequently, messages used to implement the knowledge plane can be bound to the hardware/software platform responsible for generating them.

TPM-equipped commodity PC hardware approximates our hardware security assumptions. Network devices are substantially similar to PC hardware — the primary difference is an additional high-performance switching backplane not found on most PCs, but this backplane is logically equivalent to sophisticated I/O devices, which can be attested to [82].

Technology trends suggest that future trustworthy computing platforms will be even better approximations of our hardware resilience assumptions. This suggests that device manufacturers have incentives to further improve commodity platforms. It is already possible to build highly tamper-resistant platforms, ranging from high-performance encryption of buses to protect against probing attacks [143, 85] to highly secure processors for protecting major financial and PKI transactions [22]. Thus, even today, designers of trustworthy routers can choose a level of hardware security that can support the needs of NetQuery. Even without such improvements, TPMs are becoming more deeply integrated into platforms, raising the bar for physical attacks. Should a TPM be compromised, the damage is localized, since the extracted keys can only be used to generate information attributed to that TPM.

3.2 Incomplete and inaccurate world views

The knowledge plane encodes an incomplete view of the data plane, the control plane, and the broader operating environment. Analyses must cope with the potential incompleteness to avoid deriving unsound analysis results. Poor device implementations may generate inaccurate information. Below, we present examples of potential incompleteness and inaccuracy, along with ways to ensure that analysis and device implementations can prevent or accommodate these.

3.2.1 Exogenous information

Knowledge plane incompleteness can arise because relevant system information is outside the purview of NetQuery devices. For instance, common mode failures are not always visible to NetQuery devices. Consider a pair of routers connected with multiple links. Though they appear failure independent to the devices, these links could be physically separated, or they could reside in the same undersea cable bundle and be subject to correlated failures. Similarly, independent hosts and switches in a datacenter might share a power supply and cooling system, and physical breaches or financial insolvency of the operator can disrupt many devices at once.

A fault-tolerance analysis that equates multiple links, as reported by the device, with failure independence would derive unsound conclusions. Only by including more information in the knowledge plane, such as the physical location of fibers as compiled in databases [140], can one hope to avoid such errors. The NetQuery knowledge plane can incorporate such data.

3.2.2 Inaccurate information

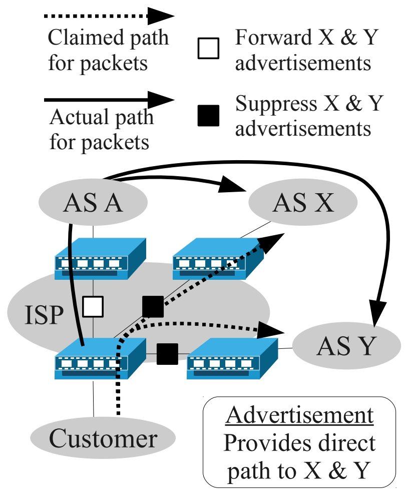
Consider a network that has routers where the forwarding and control layers behave as today, but run NetQuery and satisfy its security assumptions. This strawman device design provides weak assurances; we will show how to improve on this.

Router and data center implementations are complex. A malicious operator could well gain control of the control plane, data plane, fabric controller, or NetQuery processes, then induce them to export inaccurate facts. Trustworthy computing platforms provide isolated monitoring mechanisms [139] that operate independently of application code; using these to monitor the control plane and data plane and to export the inferred properties to the knowledge plane results in improved robustness compared with just generating properties from the full implementation.

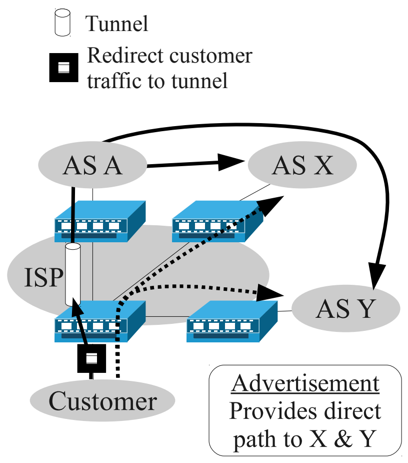
A malicious operator could introduce non-NetQuery nodes into the real network such that the nodes are not reported to the knowledge plane, yet substantially change the behavior of the network. Any actions taken based on analysis of this knowledge plane could be misguided and violate the intended policy. This attack is inexpensive to launch, since the introduced nodes can be implemented with commodity devices.

To illustrate these attacks, suppose a dishonest ISP installs additional nodes to trick NetQuery analyses into inferring a high quality network even while the actual physical network supports only a fraction of the claimed capacity, provides no redundancy, and sends customer traffic over indirect routes (Figure 3.1).[1]

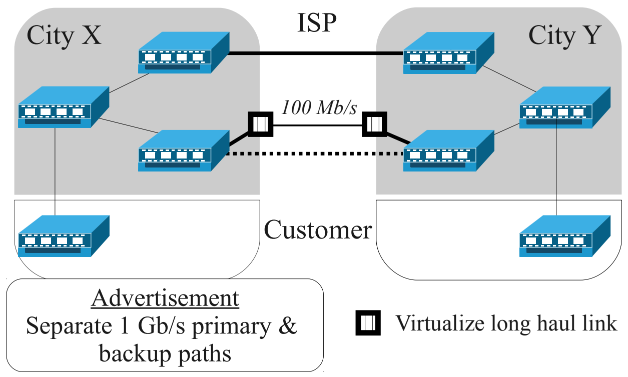
[1]Such attacks are inspired by the notion of *Potemkin villages*, in which an unscrupulous organization constructs a façade to mislead observers. The term originates from the alleged


Claimed path for packets
Forward X & Y advertisements
Actual path for packets
Suppress X & Y advertisements
AS A
AS X
ISP
AS Y
Customer
Advertisement
Provides direct path to X & Y

(a) Forge control plane messages


Tunnel
Redirect customer traffic to tunnel
AS A
AS X
ISP
AS Y
Customer
Advertisement
Provides direct path to X & Y

(b) Dissociate data plane from control plane


City X
ISP
City Y
100 Mb/s
Customer
Advertisement
Separate 1 Gb/s primary & backup paths
Virtualize long haul link

(c) Underlay virtualized network

Figure 3.1: **Attestation is not sufficient.** By using hidden nodes (denoted as squares) to control the inputs to attested devices, an unscrupulous operator can advertise false information.

In existing networks, operators have access to the keys used to secure control protocols such as OSPF and BGP. Malicious operators can use these keys to spoof routing advertisements, in a *forged control message attack*. Further, a malicious operator can also use network virtualization to hide a slower physical network or tunneling to redirect traffic along alternate paths. Such attacks embody forms of *data plane/control plane dissociation*.

practice in Imperial Russia of constructing fake villages to present an illusion of prosperity to visiting dignitaries.

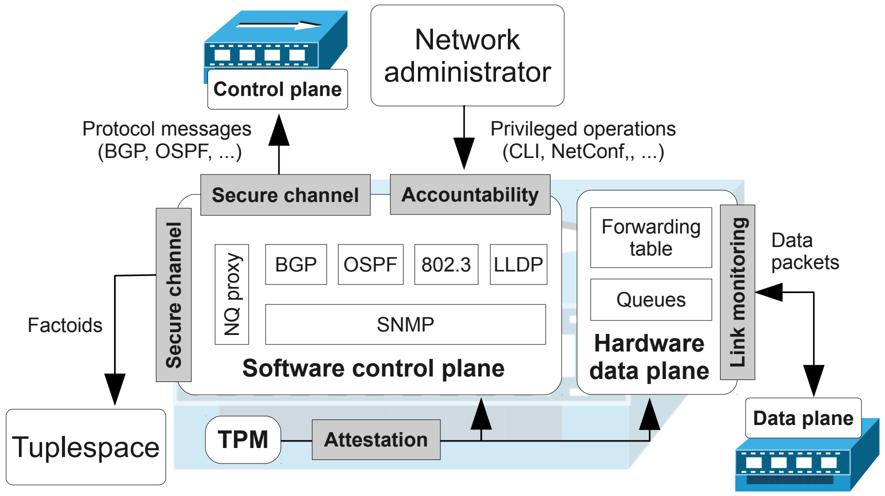


Figure 3.2: **TPM-equipped NetQuery devices.** TPM-equipped NetQuery devices bootstrap trust from embedded TPMs. Mechanisms (gray) implemented outside the TPM protect the knowledge plane against malicious operators.

In each of these cases, the knowledge plane reports that the routers are directly connected at the control or data link layer, yet they are in fact connected to an operator-controlled node. The solution is to add countermeasures to ensure that the knowledge plane and real network match. To protect against the forged control message attack, NetQuery encrypts all control messages between NetQuery-equipped devices with per-session keys known only to the devices. To protect against the dissociation attack, NetQuery devices can adopt standard solutions for monitoring the data plane for anomalies, such as trajectory sampling [55] to probabilistically detect packet redirection, link-layer encryption [5] to ensure that no control or data packets at all are redirected, and performance monitoring [24] to establish capacity bounds on every link. To achieve flexibility, extensibility, and line-rate performance, each of these mechanisms are implemented outside the TPM, with the TPM used to establish that such mechanisms are in place (Figure 3.2).

3.3 Confidentiality-preserving analysis architectures

One of the design principles of NetQuery is to leverage existing trust relationships rather than requiring new ones to be forged. This principle comes into play if we consider the problem of protecting the confidentiality of operator information, which is a central concern of operators.

3.3.1 Sanitizers

NetQuery sanitizers execute in a machine controlled by the network operator, with the assurance that the right analysis was executed, as discharged by attesting to the code that ran on the machine. This leverages a pre-existing trust relation: in the absence of NetQuery, the external party has to trust the network operator to issue a properly-derived result. Hence, a NetQuery sanitizer does not change the extant trust relationship. Rather, it simply puts the original trust assumption on a mechanically-backed basis.

An alternative would be to execute the sanitizer in a TPM-equipped machine controlled by the external party. Here, the network operator would have to ship confidential information to external machines and would have to trust those machines to not reveal this information. The trust relationship required here is substantially different from the original; operators might resist deploying this type of sanitizer. Their concern is well-founded: this architecture provides less defense-in-depth against the compromise of confidential information and there are indeed low-cost attacks that can extract confidential information from the memory of TPM-equipped machines [80].

3.3.2 Analyses spanning multiple domains

Most applications presented in this dissertation rely on multi-domain analyses that are decomposable into independent, per-domain sanitizers. Though such applications and sanitizers have been presented for the case of two domains, this structure generalizes to support any number of domains.

Not all analyses that require confidentiality have such decomposable structure; for instance, traffic engineering across multiple ASes has complex interdependencies between confidential information from mutually untrusting domains [105]. We offer here a design for an approach to such multi-domain analysis that is based on combining NetQuery with secure multiparty computation (MPC) [156].

By itself, MPC makes threat model assumptions that are difficult to satisfy and can pose risks in real-world deployments. MPC protocols do not constrain participants to be honest about their inputs and can leak information to participants that do not follow the protocol. Thus, each MPC application typically requires some application-specific security analysis of the implications of such cheating.

A NetQuery-enabled MPC implementation uses attribution and attestation to protect MPC inputs and protocol implementation. A NetQuery-enabled MPC implementation will only accept inputs to factoids from trusted principals and devices, increasing the complexity of launching attacks based on falsifying input. By attesting to the execution of a trustworthy MPC protocol implementation, NetQuery prevents participants from arbitrarily diverging from the protocol.

3.4 TPM deployment

NetQuery's flexible import policies enables applications to benefit whether or not devices are equipped with TPMs.

Deployments without TPMs For example, TPMs are not necessary in scenarios where pre-existing relationships are considered sufficient for trusting the claims of another party. Operators currently invest significant effort in establishing trust before entering peering agreements and may trust each other enough to exchange information through network diagnostic tools. This trust relationship can be represented by import policies that accept the remote operator as a root of trust in lieu of a TPM key. In NetQuery, we have the remote ISP or data center sign X.509 certificates with a self-generated key, and specify a corresponding import policy. Although this TPM-less configuration does not help with trust establishment, it still streamlines coordination and can be used to generate an audit log documenting why a claim was accepted.

Legacy devices will lack a TPM with which to generate factoids. Here, an operator can use its own self-signed key to issue statements on behalf of such legacy devices. Modern ISPs and data centers run extensive management software that collects information on the state of the network and devices within. Exporting the data from such systems into NetQuery enables even operators predominantly running legacy devices to support NetQuery applications. This approach supports a transparent transition as new TPM-equipped network devices are introduced, and operator-signed statements are subsumed by device-issued factoids.

Finally, NetQuery sanitizers can enable external applications to participate in NetQuery even when they are not equipped with TPMs. Because sensitive factoids never leave an administrative domain, lack of a remote TPM can never lead to information leakage.

External checkers Some trustworthy properties can be obtained by using TPM-equipped devices to infer or monitor legacy devices. Past work has examined how to obtain guarantees about the behavior of legacy BGP speakers by monitoring their inputs and outputs [126, 79]. Since monitors only need to inspect control traffic to infer details of BGP behavior, low cost monitoring hardware suffices to provide assurance about the behavior of expensive legacy equipment, such as high-performance routers.

3.5 Case study: NetQuery in federated networks

Depending on their configuration, administration, and provisioning, networks provide drastically different features. For instance, some networks provide little failure resilience, while others provision failover capacity and deploy middle-boxes to protect against denial of service attacks [34, 23]. Agreements between network operators often include requirements that are governed by such network features. Peering and service agreements, for example, can mandate topology, reliability, and forwarding policies, while terms-of-use agreements can mandate end host deployment of up-to-date security mechanisms. Yet the standard IP interface masks the differences between networks; each network appears to provide the same, undifferentiated "dial-tone" service. Consequently, clients and networks must resort to ad hoc techniques for advertising the quality of a network.

NetQuery can disseminate these underlying network features, thereby giving providers a channel for advertising network capabilities and enabling applications to use reasoning to find suitable networks for their requirements. This reasoning analyzes information, such as routing tables, neighbor lists, and configurations, that describe network entities, such as routers, switches, and end hosts.

3.5.1 Applications

NetQuery enables a wide range of applications based on reasoning about the properties of a remote network. Such reasoning improves the expressiveness and assurance level of inter-domain coordination.

Enforcing interconnection policies Although the level of direct interconnection between ASes on the Internet has grown substantially [154], lack of trust limits the potential benefit of this dense graph. For instance, engaging in mutual backup, wherein each AS allows the other to use its transit paths in emergencies, increases overall fault tolerance. Yet unscrupulous ASes might misuse these paths for non-emergency traffic. By checking a neighbor's BGP policies and forwarding table entries, a network can verify that backup paths are only used at appropriate times. This information is not currently available to external parties.

Verifying AS path length Since performance typically degrades as packets traverse multiple ASes, ISPs are motivated to establish peering or transit relationships to shorten AS path lengths. A small content provider or access ISP that seeks to reduce AS path length but lacks the resources to establish many direct interconnections might instead purchase service from a provider that has low AS path length connections to the desired destinations [59]. The purchaser benefits from outsourcing the overhead of managing many peering relationships and can use NetQuery to verify that traffic will be forwarded with minimal stretch. By comparison, establishing this arrangement today by using BGP-reported path information and traceroute would necessitate a large trusted computing base as well as incur the cost of active probing.

Advertising network redundancy Some networks are constructed with redundant devices and network links to increase availability. A provider with a highly redundant network can use NetQuery to advertise this fact by using a reasoning process that inspects the network topology. By comparison, it is difficult or impossible to detect redundancy with probing, since the extra resources are only visible during failure.

Avoiding rogue Wi-Fi hotspots In urban areas, mobile users are typically within range of many wireless networks [118]. Users can employ NetQuery to differentiate between these, using analysis to select networks with better security and performance. By checking for a network built from trustworthy devices and link-level encryption, a user can avoid connecting to rogue Wi-Fi hotspots [96]; by checking the capacity of the backhaul path to the upstream provider, a user can choose the best performing network in an area.

Future opportunities Many proposals for improving service discovery, such as those for bandwidth markets [146] and virtualized routing infrastructure [94], have the potential to greatly expand the set of service providers and peers available to a given ISP. NetQuery can maximize the benefits of such proposals by providing new ways to check whether a newly discovered service provider or peer is suitable.

3.5.2 Incremental deployment

NetQuery can facilitate inter-domain coordination even when only some devices and operators are upgraded to support NetQuery. Here, we describe several techniques for doing so, which are applicable to the preceding applications.

Bilateral benefits

Many claims in bilateral contracts can be computed almost entirely from factoids exported by one of the counterparties, and thus require minimal support for NetQuery outside of the participating networks. Since most Internet agreements are bilateral [59], this is a common case.

Some claims are completely self-contained within an ISP's network; these include those providing VPN service between multiple customer sites or guaranteeing an intra-domain latency or redundancy SLA. Other analyses, such as the Wi-Fi hotspot and AS path length analyzers, require a modicum of support from ASes adjacent to the Wi-Fi or transit provider. These analyses verify that the provider routes traffic to a specific destination domain as promised: for the former, to the public Internet (e.g., outside the hotspot's domain), for the latter, to the destination AS. The adjacent networks need only install NetQuery devices at the edge and assert their ownership of these device, say by signing a factoid using a AS-number certificate issued by a regional Internet registry [31]. Adjacent networks need only ensure that they deliver packets into their network as claimed by the knowledge plane. To do so, they can simply deploy a low-cost TPM-equipped host, rather than upgrade edge routers.

Islands of deployment

In other scenarios, there may be islands of NetQuery-enabled devices separated by multiple legacy devices. Islands might arise within a single provider that deploys starting at the edge of each POP or in a few POPs at a time. Islands may also be isolated by legacy devices controlled by a third party. By establishing tunnels between one another, backed by encryption and performance monitoring [24], NetQuery devices can export properties describing the intervening legacy network. Such tunnels are similar to the defenses against the dissociation attack and the preceding deployment optimization for adjacent ASes.

CHAPTER 4

PROTOTYPE AND FEASIBILITY STUDY

We have implemented a prototype of NetQuery. The core functionality is supported by an embeddable tuplespace server for building NetQuery devices and sanitizers; a C++ client library for writing NetQuery applications; NAL proof generators and checkers; and a stand-alone tuplespace server. We used these components to build the requisite devices for a federated network: a NetQuery switch for Linux, a NetQuery router adapted from the open source Quagga router [2], a NetQuery host that runs the Nexus trusted operating system, and an SNMP to NetQuery proxy (Figure 4.1). We also built a network access control (NAC) system and several network performance analyzers.

We describe, through our experience with building applications, the benefits of NetQuery-enabled analysis. We also demonstrate, through microbenchmarks of tuplespace operations and experiments and devices, that NetQuery achieves high throughput and low latency and that extending network devices to support NetQuery involves little code modification, low overhead, and low deployment cost.

All experiments used a testbed built from Linux 2.6.23 hosts equipped with 8-core 2.5GHz Intel Xeon processors and connected over a Gigabit Ethernet switch. Unless otherwise stated, all TPM and NAL optimizations from Section

Libraries	
Server & client	18,286
NAL	2,254

Data sources	
Nexus host	543
Software switch	1,853
Quagga router	+777
SNMP proxy	1,578

Applications	
Network access control	787
L2/L3 traceroute	483
Oversubscription	356
Maximum capacity	316
Redundancy	333

Figure 4.1: **Source line count for NetQuery libraries, devices, and analyses.** Router figure is the code size increase relative to Quagga. SNMP proxy supports HP and Cisco devices and exports a superset of the data for switch and router.

	Completion time (seconds)	Network cost (sent/recv'd)
L2/L3 traceroute	0.16 s	247 KB
Oversubscription	(pre-processing) 7.9 s (per switch) 0.1 s	17 MB 0 KB
Maximum capacity	0.16 s	247 KB
Redundancy	12.65 s	24 MB

Figure 4.2: **Performance of analyses on department network**. The execution time and network cost of each analysis suffices to support network management and data center SLA queries. Oversubscription analysis pre-processes the tuplespace to reduce query costs.

2.3.3 were enabled.

4.1 Applications and production deployment

Here, we outline the implementation of each application for federated networks and describe the achieved performance and operational benefits.

4.1.1 Network access control

The NAC system restricts network access only to machines that are unlikely to harm the network, such as those running a firewall and virus checkers. Such policies are widely embraced today, yet often rely solely on user cooperation. This system installs triggers on local NetQuery switches to detect new hosts, which are initially allowed only limited network access. In response to a trigger notification, the application analyzes the new host. For each policy-compliant new hosts, the application sends a configuration command to the switch to grant network access. Such policy decisions are implemented in the switch enforcer process, consisting of 787 lines of code (LOC). The NetQuery switch consists of 1,853 LOC, including a full control plane and software data plane with link-level encryption to prevent dissociation attacks. The host runs the Nexus operating system [139], which exports its full process list to a locally running tuplespace server.

To prevent leakage of sensitive user information beyond the user's computer, NAC uses a sanitizer that releases the sanitized fact $\text{CompliantHost}(H)$ if H's process list indicates that it is running the required software. NAC uses Nexus's process level attestation to verify execution of the sanitizer and tuplespace server, which run in separate processes. The proof tree for this application consists of eight ground statements: five tuplespace values, authenticated by tuplespace server MACs, and three attestations, authenticated by digital signatures.

Together, the proof generation and proof checking processes took less than 67 seconds of wall clock time, which is low compared to the long duration of typical Ethernet sessions. Digital signature verification at the enforcer dominated the cost.

4.1.2 Analysis of a production network

We deployed NetQuery on our department's production network consisting of 73 HP and Cisco L2 and L3 switches and over 700 end hosts. Using standard network management data exported by these switches, we built several analyses for detecting properties of interest to customers of cloud providers and ISPs; Figures 4.1 and 4.2 summarize the source code size and performance of each analysis. NetQuery enables these analyses to discover information that is otherwise difficult or impossible to obtain through the data plane. Our deployment relies on an SNMP-to-NetQuery proxy that periodically exports the neighbor, forwarding, routing, and ARP tables of every switch.

We implemented the following analyses, which can be used to generate remotely verifiable advertisements of datacenter network quality. These advertisements enable customer applications to pick the most appropriate network for a given workload.

L2/L3 traceroute analysis. Traceroute is widely used for diagnostics. Since standard IP traceroute returns only L3 information, it provides little information in a network composed primarily of L2 switches. We have built a NetQuery traceroute that iteratively traverses the topology graph contained in the knowledge plane, instead of using probe packets. At each switch, the analyzer performs forwarding table and ARP table lookups as appropriate to determine the next hop. To support traceroute on our network, the analysis understands many commonly used features of L2/L3 switched Ethernet networks, including link aggregation groups and VLANs. This analysis is often used as the basis for other analyses.

Over-subscription analysis computes internal network capacity and determines the ratio between the capacity of a given network link and the maximum amount of traffic that hosts downstream from the link can generate or consume. To compute the aggregate capacity across all hosts, the analysis traces through the L3 core and L2 tree, down to the leaf switches, recording every host access link. Customers with network-intensive workloads such as MapReduce can benefit from choosing datacenters that are less oversubscribed.

Maximum capacity analysis determines the available bandwidth through the Internet gateway. This analysis determines the best-case throughput between a given host and network egress point by running NetQuery traceroute to find the host to egress path, then computing the minimum capacity across the links on that path. Customers deploying public services benefit from choosing datacenters with high available capacity through the gateway.

Redundancy analysis verifies that the network is robust against network failures. We implemented an analysis that decomposes the network graph into biconnected components, which are by definition robust against the failure of a single switch. Customers that require high availability should place their nodes in the same component as critical services and the Internet gateway.

In addition to using these analyses to support external customers, the datacenter operator can use them to debug network problems. For instance, we have used these tools to help us inventory and locate network equipment and to determine the network failure modes for our research group's externally-accessible servers.

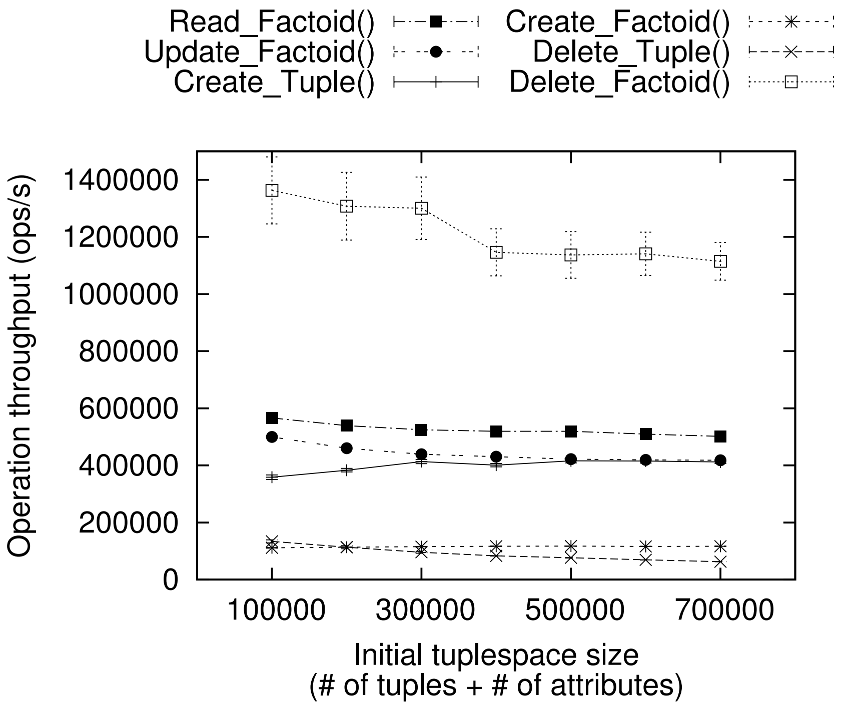


Figure 4.3: **Throughput of operations issued in bulk.** The throughput of all operations is independent of the tuplespace size.

4.2 Scalability

We evaluate the performance of a tuplespace server with throughput and latency microbenchmarks that correspond to different usages and scenarios. High throughput enables devices to initialize the knowledge plane quickly when they boot or are reconfigured. Reduced latency enables analysis to complete more quickly and limits exposure to concurrent changes to the network.

Throughput In this experiment, a traffic generator issues a sequence of requests, without waiting for responses from a tuplespace server running on a separate machine. To fully utilize processor cores available on the server, the tuplespace is distributed across eight processes.

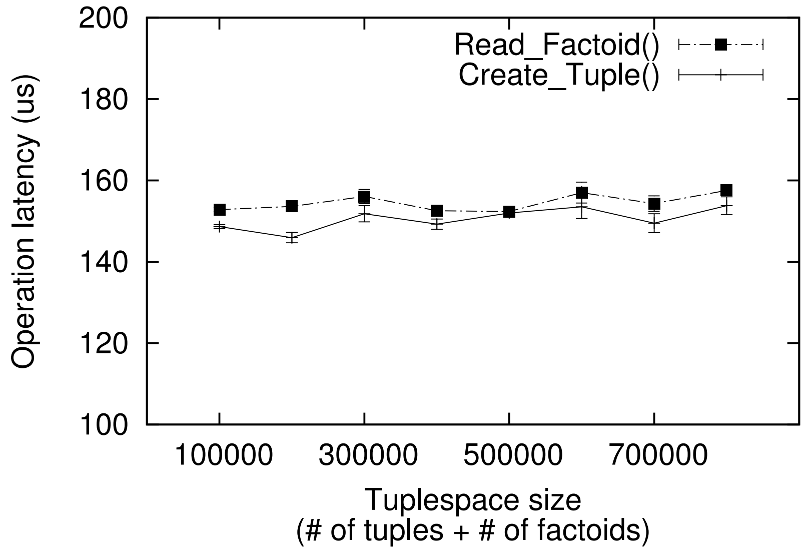


Figure 4.4: **Latency of operations issued sequentially.** The latency of all operations remains the same at all tuplespace sizes.

The results show that NetQuery can support large tuplestores and high tuplespace access and modification rates (Figure 4.3); a single tuplespace server can support more than 500,000 read and update operations per second. The throughput of all tuplespace operations is decoupled from the size of the tuplespace, with the exception of **Delete_Factoid**(). For smaller tuplespace sizes, **Delete_Factoid**() experiments complete so quickly that initialization overheads dominate overall execution time. The tuplespace server achieves this high performance because it only holds soft-state, for which a simple in-memory implementation suffices.

Latency The latency of accessing the tuplespace (Figure 4.4) affects the completion time of NetQuery applications. In this experiment, a single client issues sequential requests for the TID from **Create_Tuple**(); and for the factoid value from **Read_Factoid**(). Latency remains constant as tuplespace size increases.

SNMPv3	NetQuery (Unoptimized)		NetQuery (Optimized)	
Bulkwalk	Update	Read	Update	Read
46,572 ±300	29 ±0.06	2,723 ±30	97,422 ±2,000	94,844 ±8,000
Throughput in SNMP variables/s or NetQuery factoids/s (± 95% conf. interval)				

Figure 4.5: **Comparison of SNMP and NetQuery.** TPM- and NAL-based optimizations enable NetQuery to achieve better throughput than SNMP while providing stronger guarantees.

Cryptographic optimizations and SNMP To evaluate the overhead of accountability, we compared the performance of NetQuery to that of SNMPv3. To evaluate the benefits of the optimizations from Section 2.3.3, we measured the throughput of NetQuery with and without trusting the tuplespace server. Since the Linux SNMP server is not multithreaded, we ran both NetQuery and SNMP on a single core. We used multiple concurrent instances of `snmpbulkwalk` to retrieve SNMP server MIBs. The NetQuery experiments retrieved comparable amounts of data. SNMP and NetQuery were both configured to provide confidentiality and integrity.

When the tuplespace server is not trusted, devices sign all factoids on export and applications check signatures on all factoids on import, significantly increasing CPU overhead. When the tuplespace server is trusted, NetQuery provides better performance than SNMP (Figure 4.5). Thus, we expect that SNMP analyses that are ported to NetQuery will perform comparably, yet provide accountability.

Since TPMs and trustworthy computing can be used to establish the trustworthiness of tuplespace servers, these results show that high performance knowledge planes that support accountability can be deployed at little additional cost.

4.3 Building NetQuery devices

We built several NetQuery devices to determine the implementation and runtime costs that NetQuery adds to devices. We also show that the knowledge plane and devices can efficiently support realistic workloads.

Quagga router We modified a Quagga router to evaluate the cost of extending an existing device to support NetQuery. Only localized changes to the router's control plane were necessary to export all interface and routing table changes to the knowledge plane. NetQuery-Quagga interposes on Quagga's calls to `rtnetlink`, the low-level interface to the in-kernel dataplane, and translates all relevant requests, such as changes to the forwarding table and NIC state, into tuple updates. In total, only 777 lines of localized changes were needed, out of a total code base of 190,538 LOC.

Initialization Quagga is a demanding macrobenchmark that exports significant amounts of state during operation. To demonstrate that routers can efficiently shed this data to a tuplespace server, we measured the initialization and steady state performance of the router. We used a workload derived from a RouteViews trace. The Quagga router, tuplespace server, and workload generator ran on separate machines. To demonstrate that NetQuery can efficiently import bulk data, we measured the completion time to load a full BGP routing table (268K prefixes) and the resulting tuplespace memory footprint.

Upon receiving routing updates, the BGP router downloads a full forwarding table to the IP forwarding layer. Added latency could affect network availability. Without NetQuery, an update of the full table took 5.70 s; with NetQuery, it took 13.5 s. Our prototype blocks all updates while waiting for NetQuery; a production implementation can eliminate this dependency by updating the knowledge plane in a background process.

Exporting the full forwarding table to the tuplespace server required 62.8 MB of network transfer and 10.7 MB of server memory to store the table. Though full updates are the most intense knowledge plane update workload, only a modest amount of hardware is needed to support them.

Steady state To demonstrate that NetQuery routers perform well in steady state, we evaluated the router against a workload derived from RouteViews update traces. The workload generator batched updates into one second buckets, and submitted them as bursts. The experiment recorded the time needed to commit the resulting changes to the IP forwarding tables and to the knowledge plane. NetQuery increased the median completion time to 63.4 ms, from 62.2 ms in the baseline server. Thus, the NetQuery router reacts almost as quickly as a standard router, minimizing the disruption to forwarding table updates. NetQuery required only 3 KB, 92 KB, and 480 KB to transmit the median, mean, and maximum update sizes; thus, any server configuration provisioned to support initialization load can also support steady state load.

Convergence time NetQuery does not impact eBGP convergence time: in eBGP, route propagation is governed by a thirty second Minimum Route Advertisement Interval, which exceeds the latency of exporting a full forwarding table update to NetQuery.

To measure the impact on IGP route convergence, we simulated the update traffic from large correlated link failures on the Sprint RocketFuel topology. This topology consists of 17,163 Sprint and customer edge routers. We converted the simulation trace into a POP-level NetQuery workload, which we fed to a single-core tuplespace server.

We measured for each run the convergence time after failure. For the five largest POPs, consisting of 51 to 66 routers, and link failure rates of up to 0.05, the mean and median increase in update completion times were less than 0.24 s and 0.14 s, respectively. Thus, networks can deploy NetQuery while achieving sub-second IGP convergence time, which is the desired level of performance for operators [64].

CHAPTER 5

INFRASTRUCTURE SUPPORT FOR FEDERATED CLOUD COMPUTING

Cloud computing is becoming increasingly important for building many types of applications and Internet-facing services [21]. Traditional computing infrastructure poses significant costs to deploying scalable, wide area services, with each service duplicating the effort of provisioning physical facilities to provide enough computational resources and designing the abstractions necessary to scalably manage and exploit these resources.

By comparison, cloud computing enables substantially better application agility in acquiring and coordinating computational resources. Cloud computing providers take on the responsibility of designing and building scalable computational infrastructure. The cloud infrastructure provides standardized interfaces for provisioning resources for applications: applications can request computational resources in an on-demand, pay-as-you-go fashion, and can readily move across different cloud providers to exploit geographic diversity, maximize availability, and reduce costs.

The fungibility and portability of computational tasks and resources enables new types of market transactions. For instance, specialization generates efficiency gains in cloud computing. By aggregating many customers and tasks into the same data center, cloud services [15, 110] can drive down costs through economies of scale and by optimizing the size and geographic placement of data centers [70]. Given that cloud applications can move computation to lower-cost sites, cloud computing can be simultaneously profitable for operators while reducing costs for customers [21].

Cloud computing can also transfer risk between different participants in the global computing infrastructure [21]. Traditional computational infrastructure require developers of new services to provision servers and network capacity in anticipation of future demand levels. But making accurate demand predictions for novel applications can be challenging, and over- and under-predicting demand can both incur considerable cost. Given these trends, the on-demand, pay-as-you-go cloud service model is favorable because it shifts responsibilities and risks associated with capacity planning to the cloud provider. Since cloud providers run many different applications on their infrastructure, they can statistically multiplex a common pool of reserve capacity for servicing peaks to reduce the costs and risks associated with misprediction.

While running VMs and tasks from multiple tenants in a common set of cloud data centers contributes to the aforementioned benefits, such heterogeneous workloads can also pose a resource scheduling challenge. Since the VMs and tasks sharing a cloud data center come from unrelated customers, they are largely uncoordinated and mutually untrusting. Thus, the potential for network performance interference and denial of service attacks is high. As a result, performance predictability is a key concern [47] for customers evaluating a move to cloud datacenters. Though existing cloud data centers provide many mechanisms to schedule local compute, memory, and disk resources [41, 75], their mechanisms for apportioning network resources and isolating applications from different tenants fall short, resulting in inefficient network utilization and increased vulnerability to attack.

In principle, fungibility and portability could enable participants in a global cloud computing infrastructure to offer, repackage, or trade computational capacity, enabling more efficient infrastructure utilization. Tenants typically have performance, reliability, and confidentiality requirements regarding the computational environment in which their applications execute and can only run applications within suitable data centers. But the existing cloud infrastructure provides few mechanisms with which such requirements can be verified, with existing mechanisms based largely on ad hoc reputation. Thus, the market is far from flat: the overhead of establishing reputation and trust benefits incumbents and new service providers cannot convince users that they can satisfy application requirements without first incurring the cost of building sufficient reputation. As a result, today's cloud computing infrastructure is minimally federated and is not well-suited for building an open market for cloud computing resources [98].

The remainder of this chapter highlights the problems of existing resource allocation and security mechanisms and introduces novel data path techniques for implementing better mechanisms. It also shows how NetQuery can be used to facilitate coordination across a federated cloud computing infrastructure. Taken together, these contributions greatly expand the kinds of guarantees that cloud applications can rely on.

5.1 Establishing new types of guarantees

Compared with workloads found in the Internet and traditional data centers, typical cloud computing workloads pose substantial challenges to building cloud data center networking guarantees, such as granular network resource allocation and security mechanisms. Individual malicious nodes on the Internet can generally consume a much lower fraction of core network bandwidth than individual malicious VMs can consume in the data center. Since traditional data centers generally execute fewer, less diverse tasks from a single administrative domain, they can more readily mitigate misbehaving and interfering tasks.

Cloud data centers often rely on the network resource allocation and security mechanisms inherited from preceding deployment scenarios such as the Internet and traditional data center equipment. Since cloud data centers have substantially different engineering parameters and workloads from these scenarios, such mechanisms are often ill-suited for use in cloud data centers. For instance, end host mechanisms, such as TCP congestion control, are scalable and widely deployed, yet are not robust against misbehaving tenants. Switch and router mechanisms, such as reservations, can provide a measure of robustness, but are not scalable to the number of VMs and tasks found in the typical data center and may leave the network underutilized.

Cloud computing applications are deployed on cloud computing stacks, such as those based on low-level hardware virtualization (Infrastructure-as-a-Service) or high-level software platforms (Platform-as-a-Service) [147]. Cloud computing data centers often employ tight integration between all components in the data center, including end hosts and the network. They are orchestrated by a centralized fabric controller responsible for allocating host and network resources for executing tenant applications [112, 114, 98]. This integration provides opportunities for building new abstractions and algorithms for controlling resource allocation. Combined with the availability of open platforms [109] and customized, high-performance switches based on merchant silicon [60], it is feasible for data center operators to deploy custom, clean slate solutions spanning the entire packet forwarding pipeline, from end host network stack to physical network switches [11].

5.1.1 Trusted packet processors

This degree of control over data centers can be used to implement *trusted packet processors*, which provide programmable data plane processing that is isolated from untrusted tenant-supplied code. Trusted packet processors can be used to build many new data center networking guarantees; by attesting to the programs executing on each trusted packet processor, a data center can convince remote parties that a given data center networking guarantee is installed.

To support many applications, trusted packet processors should provide general programming models and extensive coverage. Enabling applications to apply custom processing to specific subsets of traffic allows targeted policies, while enabling applications to apply custom processing at different locations in the network provides better observations of network state. To have practical benefit, any new cloud data center networking guarantees would need to scale to support the large number of end points, high churn, and high data rates seen in typical cloud workloads. Since cost and efficiency are important considerations in cloud data centers, the implementation techniques should be compatible with the technical limitations of typical data center networking hardware and software.

5.1.2 Trusted packet processors at the network edge

End host virtual switches and network stacks can provide trusted packet processing coverage for the network edge, with ample processing capacity for the comparatively low bandwidth demands at such locations. Virtualization and hardware privilege separation provide isolation from tenant code.

Since such trusted packet processors execute on the end host's primary CPU, they can support a general programming model that can run user-specified packet processing code on any traffic leaving a node. Trusted packet processors based in end hosts can also be deployed with only software changes, which are straightforward to integrate with the open, extensible architectures of typical virtual switches and network stacks and can leverage the centralized software provisioning mechanisms of a cloud data center.

When fully deployed, end host-based trusted packet processors can observe and manipulate all traffic generated by internal end hosts; such traffic typically dominates overall data center traffic since intra-datacenter links have substantially higher bandwidth than Internet access links. Should coverage over the remaining externally-generated traffic be needed, border routers can be augmented with middleboxes that provide the same trusted packet processing.

Applications

The comprehensive coverage that trusted packet processors at the network edge provide can be used to build applications that employ end-to-end control algorithms and policies for resource allocation, performance isolation, and reachability isolation for gaining control of the heterogeneous, untrusted mix of applications running in a typical cloud data center.

Seawall: Network capacity allocation and performance isolation To provide performance guarantees for applications and protect against malicious or selfish tenants, a cloud data center should be able to control how network bandwidth is shared among multiple tenants, regardless of what traffic the tenants may send.

Existing adaptive edge-to-edge techniques, such as TCP congestion control (or variants such as TFRC and DCCP), are scalable and achieve high utilization. These are widely deployed, scale to existing traffic loads, and, to a large extent, determine network sharing today via a notion of flow-based fairness. However, TCP does little to isolate tenants from one another: poorly-designed or malicious applications can consume network capacity, to the detriment of other applications, by opening more flows or using non-compliant protocol implementations that ignore congestion control. A tenant can use the extra bandwidth selfishly or use it to interfere with others on the shared links, switches or servers. Thus, while capacity allocation using TCP is scalable and achieves high network utilization, the achieved results are not robust against variations in tenant communications patterns.

By comparison, existing switch and router mechanisms (e.g., CoS tags, Weighted Fair Queuing, reservations, QCN [117]) are better decoupled from tenant misbehavior. However, these features are of limited use when applied to the demanding cloud data center environment, since they cannot keep up with the scale and the churn (e.g., numbers of tenants, arrival rate of new VMs), can only obtain isolation at the cost of network utilization, or might require new hardware.

This thesis proposes a new end-to-end control mechanism for allocating network capacity. Called *Seawall*, this mechanism is built from trusted packet processors at the edge and enables administrators to prescribe how their network is shared. It allocates network capacity and provides performance isolation guarantees irrespective of tenant traffic characteristics such as the number of flows, protocols or participating endpoints. Here, we outline the general properties of Seawall and defer to Chapter 6 a full description of the design and implementation of Seawall and associated performance evaluation.

Seawall provides a *network weight* abstraction for defining network allocation policies. Given a network weight parameter for each local entity that serves as a traffic source (VM, process, etc.), Seawall ensures that along all network links, the share of bandwidth obtained by the entity is proportional to its weight. To achieve efficiency, Seawall is work-conserving, proportionally redistributing unused shares to currently active sources.

Beyond simply improving security by mitigating DoS attacks from malicious tenants, per-entity weights extend existing differentiated provisioning models, where tenants can pay more for VMs with larger share of local resources, to cover bandwidth allocation. Per-entity weights can also enable better control over infrastructure services. Data centers often mix latency- and throughput-sensitive tasks with background infrastructure services. For instance, a search cluster requires low latency for index queries from the users, which contends with the massive bandwidth requirements of periodic index updates. Similarly, customer-generated web traffic contends with the demands of VM deployment and migration tasks. Per-entity weights serve as a mechanism for avoiding disruption between these concurrent workloads.

Seawall achieves scalable capacity allocation by reducing the network sharing problem to an instance of distributed congestion control. The ubiquity of TCP shows that such algorithms can scale to large numbers of participants, adapt quickly to change, and can be implemented strictly at the edge. Though Seawall borrows from TCP, Seawall's architecture and control loop ensure robustness against tenant misbehavior. Seawall uses a shim layer at the sender that makes policy compliance mandatory by forcing all traffic into congestion-controlled tunnels. To prevent tenants from bypassing Seawall, the shim runs in end host-based trusted packet processors. Simply enforcing a separate TCP-like tunnel to every destination would permit each source to achieve higher rate by communicating with more destinations. Since this does not achieve the desired policy based on per-entity weights, Seawall instead uses a novel control loop that combines feedback from multiple destinations.

Reachability isolation In a cloud data center, a malicious tenant can send high data rate traffic to the VMs of an unrelated victim tenant located in the same data center. This unwanted traffic poses a threat to performance guarantees and security, with the malicious tenant launching DDoS attacks or compromising the VMs of other tenants that are sharing the network.

By supporting reachability isolation policies, a cloud data center can impose restrictions on what VMs are permitted to communicate. For instance, a data center can restrict tenant VMs to exchanging packets only with other VMs belonging to the same tenant or with shared services provided by the infrastructure.

In a sense, we would like to to place each tenant on a private network. However, achieving this by placing each tenant's VMs in a dedicated Ethernet VLAN does not scale due to limits on the number of VLANs in one L2 domain (no more than 4096). Enforcing isolation by placing ACLs (e.g. VLAN or IP-based) in switch hardware will quickly exhaust TCAM storage, as the number of ACLs scales at least linearly with the number of VMs whose traffic can transit the switch. Scaling up the supported policy sizes requires upgrading to more expensive switches; such switches are prohibitively expensive at the top-of-rack (ToR) level.

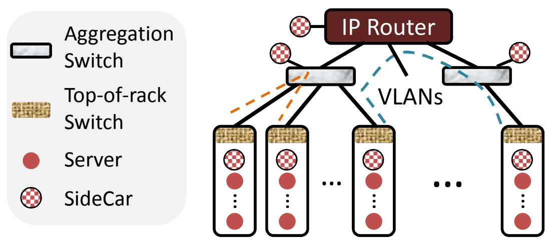


Figure 5.1: **Extending a datacenter network with sidecars.**

Likewise, supporting new types of policies, such as tenant- or user-based ones, can require replacing all switches [43].

End host trusted packet processors can be used to enforce reachability policies at a fine granularity. Such policies can precisely specify the set of VMs that can communicate, regardless of policy size, network topology, or traffic matrix. The specified policies cannot be bypassed, since trusted packet processors execute in isolation from tenant-specified code. Moreover, by running packet filters at the source rather than at the destination, the system prevents malicious tenants from wasting network bandwidth on disallowed traffic.

5.1.3 SideCar: In-network trusted packet processors

With visibility and control beyond the network edge, applications can implement a wider range of network guarantees. Adding trusted packet processors at the ToR, core, and aggregation switches enables new functionality. With improved visibility, applications can collect utilization and congestion information down to the granularity of individual network links, which they can use to improve congestion control. Applications can also customize forwarding abstractions by running code in these in-network trusted packet processors.

In-network trusted packet processors also improves security over networks that rely exclusively on end host packet processing. Existing networks provide defense-in-depth: even when end hosts are compromised, the fabric controller can still use switches to observe and control the network, detecting and isolating suspicious end hosts as needed. The attack surface for this extra layer of protection is small, since end hosts interact with switches only through the packet forwarding interface.

By comparison, suppose a malicious tenant VM compromises its local hypervisor and disables end host packet filtering; such VMs can launch attacks on VMs from other tenants. By augmenting end host packet processors with in-network packet processors, data centers can relax the trust dependency on end host packet processors, thus providing defense-in-depth and reducing TCB size.

In-network packet processors also enable safe use of common performance optimization such as direct I/O, in which latency- and throughput-sensitive tenants are permitted to send traffic directly to the network. Such hosts cannot support trusted packet processing, since their data path bypasses the virtual switch.

Challenges, programming model, and architecture

It is costly to build in-network trusted packet processors that can handle all traffic traversing the aggregation and core layers. Although fully-programmable routers with software forwarding paths that can perform at the necessary data rates have been proposed [53, 81], they incur substantially higher hardware and power costs than traditional routers that employ fixed-function fast forwarding paths. Thus, routers based on software forwarding paths are likely untenable in cost-conscious, production cloud data centers.

Instead, this thesis proposes an alternative programming model that provides full coverage of network topology from network edge to core, with small incremental cost over existing switches. Called SideCar, this programming model reduces hardware cost by limiting trusted packet processors to handling just a fraction of traffic, rather than all traffic. Despite this limitation, the processing model can support many new applications.

SideCar's trusted packet processing executes in sidecar processors, which lie off the fast path for packet forwarding (Figure 5.1). This execution model is an extension of the slow-path packet processing used by all modern switches. To achieve cost and performance requirements, the majority of traffic on a switch is processed by a comparatively inflexible data-path. Only packets that require more sophisticated processing, such as control traffic and packets that induce forwarding state changes, are forwarded to a programmable control processor.

In typical data center switches, the redirection mechanisms provide limited flexibility for redirecting traffic for special processing. Since switches use TCAMs to match traffic, they can only support a small number of packet match rules of limited complexity, and scaling up requires costly switch upgrades. Instead of relying on switches to perform classification, SideCar delegates classification to end hosts, which can use software-defined rule sets that are larger and more expressive that can scale to the low data rates at the edge.

This pre-classification at end hosts facilitates scalable packet redirection or copying to sidecars that precisely intercept packets requiring special processing. Called *steering*, this process has two variants; the underlying redirection primitives are simple and widely supported at line rates by existing hardware (e.g., VLAN tagging [111], IP-IP encapsulation [71]). In *marked* steering, each switch redirects all packets with a special mark to the sidecars. Packets can be *direct* steered by sending packets to a specific sidecar. Marking and direct steering can be used both at the datacenter edge, i.e., shim layer in the host network stack or hypervisors, and between different sidecars to implement new types of forwarding schemes.

By itself, pre-classification at the end host neither protects against compromised end hosts nor supports direct I/O optimizations. Thus, SideCar augments steering with sampling to enhance security and performance. Rather than fully trusting the end host software stack to perform all critical packet processing, which introduces potential vulnerabilities, or doing all such processing in sidecars, which are trustworthy but may not have enough computational power to do so, applications can use the available sidecar capacity to spot check the work of the end host, providing a probabilistic bound on misbehavior.

The choice of what to use as sidecars ranges from commodity servers to RouteBricks or PacketShader class servers. This choice depends on the the volume of traffic transiting the switch that the sidecar is connected to and the demands of the application, i.e., what fraction of packets to observe and what to compute per packet.

Sampling and Steering

To provide all aforementioned properties, packet sampling must not have dependencies on any untrusted layers; sampling that is implemented by relying on untrusted layers to mark a subset of packets improves only scalability. Sampling primitives that are built into switches are free from such dependencies: such primitives, like NetFlow, sFlow, and sampled port mirroring, are supported in nearly all switches. Each of these redirect a sample of traffic to a pre-configured monitoring node NetFlow [45] collects aggregate flow-level statistics for packets that it samples. Given a sample rate, sFlow [122] picks packets uniformly at random and collects their headers for analysis; sFlow is available on even entry-level ToR switches [83]. Sampled port mirroring uniformly captures traffic matching a designated TCAM pattern.

Port mirroring can sample full packets on switch ports and forward them to another port on that switch. Switches that implement packet mirroring in fabric hardware are limited only by the bandwidth of the outgoing port and can deliver packets with low latency. A high performance sidecar, say one connected to a ToR switch with comparatively low aggregate bandwidth, may take in all packets from the switch and itself implement more precise statistical sampling techniques such as trajectory sampling.

Either a switch or an end host can determine which packets to steer to a sidecar. There are a few ways to achieve steering. Analogous to mechanisms such as MPLS or DOA [129, 151], packets can be directly steered by tacking on an outermost routable header with the address of a target sidecar. Steering can use MAC-in-MAC encapsulation or IP-in-IP encapsulation to specify the target sidecar's address. To implement marked steering, switches are configured to

Type of sidecar	Cost	Appropriateness		
		Location	Cost / port †	Application
8-core Commodity Server, 1 or 10Gbps NIC	$2800	@ ToR switch	$70	can process 1.25-4% of ToR's traffic
		@ Agg switch	N/A	unsuitable, can only process < .1% of traffic
RouteBrick / PacketShader server, up to four 10Gbps NICs	$9000	@ ToR switch	$14	can process 50% of ToR traffic
		@ Agg switch	$8	can process ~ 2.6% of Agg's traffic
n-port RouteBrick / PacketShader router	$9000**n*	@ Agg switch	N/A	can process all Agg traffic, L3 replacement

† Cost per 1 GbE ToR port, to process at least 2.5% traffic at indicated location, assuming 2:1 over-subscription.

Table 5.1: **Comparison of implementation options for sidecars**. The cost and suitability of each option varies depending on network location and application demands.

redirect to a sidecar traffic that has a specific VLAN tag (works in L2) or type-of-service tag (works in L2 or L3) or a specific MAC or IP destination address. The edge or downstream sidecars can mark packets for processing at the next sidecar on the path.

Choosing an appropriate sidecar

Table 5.1 compares a few options for sidecars on their cost, as of July 2010, and appropriateness for network locations and application demands.

Let us consider the cost of using commodity server hardware as sidecars. An 8-core compute node with one 1 GbE link costs $2000. Swapping in an optical 10Gbps NIC increases the price by about $800. A ToR switch may have 40 1 GbE downward-facing ports attached to nodes in the rack and four 10 GbE upward-facing ports attached to aggregation switches. ToR switches are typically low-end switches (limited ACL, VLAN, multicast capability, no L3 support) and cost \$3000, for a per-node cost of \$75. Note that, due to differences in volume and market segment, switches and sidecars can have different levels of markup; thus, comparing list prices only approximates a comparison of true costs. RouteBricks and PacketShader-class hardware have higher end processors and I/O interconnects costing \$4000-7000; the cost of optics for each of four 10GbE NICs brings up the cost to about \$6000-9000. The performance of commodity servers is set to improve since low end processors will soon ship with better I/O interconnects and on-die GPUs.

A single commodity server is suitable as a sidecar at the ToR switch. It can process sizable fractions of the traffic traversing the ToR switch and perform moderate computations per processed packet. The per-node increase in cost due to the sidecar is about \$50 for a node with a 1 GbE link that can process 1.25% of the packets through the ToR switch (\$70 for a node with a 10 GbE link that can process 4%[1]). The final cost is comparable to that of a high-end ToR switch; SideCar's programmability enables it to match or exceed some of the features of such switches.

Note that a number of optimizations are possible. Applications that only require lightweight packet processing or only monitor a low volume of traffic allow sidecars to be shared between different switches, while more intensive workloads can use higher end sidecars. Performance estimates from RouteBricks and PacketShader suggest that one server can support up to 35-40Gb/s today. A PacketShader blade split across 16 different ToRs can process 3.1% of the packets at each of these ToR's and costs \$14 per port, while one split across 4 different

[1]A commodity server with lower bandwidth I/O interconnects cannot process more than 3-5Gbps [53]

ToRs can process 12.5% of the packets for $56 per port. Both configurations are more cost-effective than using commodity blades.

An important special case is that of a non over-subscribed datacenter network. When one of the many recently proposed datacenter topologies (VL2, fattree, bcube, dcell) eliminate over-subscription in the core of the network, many applications such as monitoring, multicast support and controlling network bandwidth allocation only require programmatic control outside the network core, i.e., at the ToRs.

If necessary, however, RouteBricks or PacketShader class server can play the role of a sidecar at aggregation and core switches. Assuming a 1:2 over-subscription ratio, the cost increase is $8 per server to be able to process 2.6% of the traffic entering the aggregation or core switch; this includes the $16,000 cost of consuming four 10 GbE ports on a 160-port aggregation or core switch. The cost falls linearly as over-subscription factor increases. In non-oversubscribed topologies that use more switches, the per-port cost increases linearly with the number of additional aggregation or core switches needed to support the topology.

Applications

We show how SideCar facilitates novel solutions to four pressing problems in large-scale cloud datacenters and in enterprise networks.

Enhanced Seawall control protocol In the baseline version of Seawall, the congestion control loop relies solely on end-to-end congestion signals. Inferring congestion signals in an end-to-end fashion results in slower control loop response and causes suboptimal performance when all traffic is bundled through a small number of congestion controlled tunnels.

SideCar improves edge-based congestion control by using in-network trusted packet processors to provide XCP [93]-like explicit feedback. By providing explicit feedback, in-network trusted packet processors on the path enable the send hypervisors to quickly yet stably converge to their appropriate shares. Specifically, the trusted packet processors along the path sample uniformly from all packets to estimate the amount of spare bandwidth available on the bottleneck link and the identities of the hypervisors that are using the link. By passing these values back to the send hypervisors, each hypervisor can make a judicious choice of adapting its share. The increase and decrease rules, while analogous to XCP, have to be modified since only a sample of all packets are observed and since feedback is not issued per-packet.

Reducing TCB size for reachability isolation The edge-based reachability isolation system trusts end hosts to implement packet filtering in accordance with the network's reachability policy. Should an end host be compromised, a malicious VM may be able to bypass the policy. To prevent this using only edge-based packet processing, virtual switches can implement packet filtering to reject any received packets that violate the policy. While this approach prevents unwanted traffic from reaching the VMs, an attacker can force the receive virtual switch to waste CPU cycles processing these packets or burden the network with unwanted traffic. One might consider proactively blocking such traffic using the receiver virtual switch to detect violations on behalf of the fabric controller; in response, the fabric controller could quarantine the offending VM. However, this introduces another potential denial-of-service attack, wherein a compromised node falsely accuses a well-behaved VM of launching an attack.

SideCar leverages sampling to achieve, for significantly lower cost, the flexibility of hypervisor filtering while providing defense-in-depth and aggressive containment of attacks. A probabilistic guarantee of detecting policy violations, combined with an aggressive mechanism to contain attackers, suffices to limit the damage due to an attacker. Though this probabilistic guarantee temporarily permits some packets that switch-based filtering would reject outright, it can implement finer-granularity policies and scale to larger numbers of VMs.

SideCar uses sample-based auditing to enforce reachability isolation, achieving, for significantly lower cost, the flexibility of hypervisor filtering while providing the same level of DoS protection as switch-based filters. SideCar switches do not filter packets; rather, they expect end hosts to filter packets, but execute trusted packet processors that check a uniform sample of packets against the reachability policy. By itself, sampling provides only reachability isolation detection. To provide reachability isolation, we combine this with a strong response to policy violations. When violations are detected, the fabric controller configures the ToR to revoke the sender VM's access to the network.

The detectors, on account of being physically separate sidecar processors, are harder to compromise. Further, we use address spoofing prevention mechanisms at the edge (i.e., in ToR switches and hypervisor virtual switches) to prevent an attacker from spoofing unwanted packets as if they were coming from someone else. Thus, an attacker cannot cause SideCar to revoke access of innocent nodes.

Sampling provides a probabilistic guarantee – with high probability, SideCar detects violations within a small number of packets. Suppose p is the probability of sampling a packet. Then the probability of detection after sending k policy-violating packets is $1 - (1 - p)^k$. When multiple in-network trusted packet processors lie on a path from the source to the destination, there are further detection opportunities. Assuming independent sampling at m SideCar switches along the path, the detection probability increases to $1 - (1 - p)^{m \cdot k}$. Given a sampling probability of 1% and two SideCar switches, the detection probability is 99% after 227 packets. [2]

This packet limit applies to all destinations from a given VM; each VM can incur only a limited number of violations. This bounds the total packet leakage between colluding hosts on isolated networks and bounds the amount of host CPU and network capacity consumed on filtering unwanted packets that violate the policy.

VM migration and churn When VMs move or new VMs are created or removed, the set of applicable policies changes. The fabric controller, which is responsible for coordinating such actions, communicates the change to sidecars. For a short time after a policy change, stale packets in the network may be mistaken for violations. To guard against such false positives due to race conditions, SideCar accepts packets that conform to the old policy up to a timeout period after the change.

Cost comparison with edge-only and switch-only approach By retaining filters at the destination virtual switch, the probabilistic guarantee of the sample-based approach can be converted to a deterministic one since any packet that slips past will be dropped. This incurs the same end host filtering overhead on

[2]increasing from 99% after 458 packets with just one switch.

conforming traffic as the edge-only approach yet provides substantially better protection against denial of service attacks. Compared with using large switch TCAMs, the sample-based approach supports more expressive policies and can be retrofitted without replacing switches en masse.

More expressive policies SideCar allows for highly expressive policies and scales to the size and churn in cloud datacenters. Similar to Berkeley Packet Filters [108], SideCar's policies in software can be specified over hosts, ports and applications. Checking for policy violations is embarrassingly parallel. PacketShader reports a rate of $>$ 10 mpps with two GPUs on similar rules. Further, much work to speed up filtering large rule sets in the context of Berkeley packet filters [27] and IDS boxes can be leveraged in software.

Scalable, programmable multicast Multicast can improve the performance of many abstractions for building large scale systems, such as consensus [36], data replication [38, 67] and mass VM start up [99]. However, the use of native IP multicast has been limited in enterprise datacenters due to concerns about its congestion stability and security [148]. Likewise, cloud providers typically do not expose multicast to their tenants.

End hosts steers packets needing multicast support to a SideCar switch on the path. SideCar maintains multicast state in the sidecars. For each group whose traffic transits through a SideCar switch, the sidecar of that switch maintains a list of the switch ports having participants in the group. Upon receiving multicast packets, a sidecar replicates the packet as necessary and forwards it out the other ports.

Multicast primitives can be built using SideCar that avoid the security and congestion implications of native IP multicast [50]. Such out-of-band support for multicast is similar to Application Layer Multicast [42] but is better performing due to fewer needless copies of packets and shorter paths. Congestion control or back pressure can be done in software to improve stability.

Supporting multicast is feasible with SideCar. Even commodity servers can handle table lookup and packet replication for up to modest volumes of multicast traffic. When the fan-out of a group is large, SideCar leverages support for local multicast groups in switch hardware. By constructing a local multicast group entry consisting of the outgoing ports that this traffic should leave on, the sidecar needs to transmit just one copy of the packet and defer replication to the data plane in the switch.

Preserving hypervisor policy control under direct I/O To improve network performance in virtualized environments, direct I/O from guest VMs to the NIC [54] has been recently standardized. Direct I/O avoids the overhead of passing packets through the hypervisor. However, it comes at the cost of losing the policy control (e.g., filters, rate limiters) that is currently done in the virtual switch.

SideCar provides a way to restore the policy control of the virtual switch without waiting for network usage policy enforcement in direct I/O to become standardized, implemented, and available. Consider the example of rate limiting a VMs traffic to the network. SideCar achieves this by asking the guest VM to limit its traffic. However, there is no guarantee that the guest VM, which can be arbitrarily modified by the tenant, will do so. Hence, a SideCar switch at the ToR upstream of the guest samples all traffic leaving the ToR. By sampling uniformly at random, SideCar can project from the proportion of the guest's traffic seen in the sample to check that the VM is within limit.

Consider another example of controlling a VM's resource utilization on a storage area network (SAN) that provides many hosts and VMs with access to a common set of disks. A VM's seek load and bandwidth consumption can impact the throughput of other VMs accessing disks on the same SAN [75]. SideCar can enforce policies on SAN access without interposing on all traffic. Instead, SideCar expects the guest to mark and steer disk command packets to a sidecar that can track these metrics. To prevent a guest from deflate its request rate by only steering some of its command packets, SideCar uses sample-based spot checking to probabilistically bound the number of request packets were steered incorrectly.

In this discussion, we assume use of a datagram-based SAN protocol, such as FCoE [7], with a software-based initiator that sends requests over a direct I/O NIC. SideCar expects the guest VM to limit its disk access rate. Controlling SAN access requires a different verification algorithm from network bandwidth, since both seek time and throughput are bottlenecks on mechanical drives. Since disk schedulers commonly use the rate of I/O operations and latency to approximate seek overhead, SideCar measures these as well as throughput. To do so, only SCSI command packets (e.g., block read and write requests and their status codes, but not their payloads) are steered to and inspected by SideCar. SideCar parses the commands to measure the number, transfer size, and latency. Inspecting only command packets and not data packets is more efficient since mechanical drive arrays only support 100s-1000s of I/O operations per second, yet can easily saturate a SAN link with data. To avoid having to modify to the NIC or ToR to recognize command packets, the guest VM is responsible for steering the command packets. SideCar uses uniform random sampling to enforce proper steering with spot checking. SideCar parses every sampled packet to verify that FCoE SCSI control packet are properly steered. This check yields a probabilistic guarantee: while some control packets may be improperly steered, long-term cheating will be detected with high probability. For a sample rate of 1%, the probability of evading detection drops to less than 0.1% after issuing 172 I/O operations (each I/O operation generates four control packets). Since this detection generates no false positives, SideCar can terminate the VM or raise an alarm after detecting any violations. Thus, a malicious or selfish VM is limited in the amount of damage that it can do.

5.2 Building a knowledge plane for federated cloud computing

For reasons of service quality, reputation, and regulatory compliance, cloud tenants can require performance, reliability, and confidentiality guarantees from the resources delivered by cloud providers. For instance, the payment processing and health care industries mandate specific security requirements on the computing infrastructure of participants [9].

But the current cloud infrastructure is ill-suited for providing such assurances and exports only limited interfaces for coordinating between (1) cloud providers and tenants and between (2) cloud providers and other cloud providers. Today, such guarantees are exchanged in an ad hoc fashion and are often based on a provider's reputation. Reputation can be difficult to acquire and poses a market barrier that is biased towards established providers[3]. Despite the theoretical fungibility and portability of applications built using cloud computing abstractions, the potential gains of diversifying to new providers or constructing new cloud service offerings are sometimes dwarfed by the economic transaction costs of determining compliance with requirements.

For instance, suppose a large, reputable company has excess computational capacity in a private cloud used to support sensitive internal applications. Rather than allowing this equipment to sit unused, wastefully depreciating in value, the company could recoup costs by selling spare capacity to the public cloud [57]. But the cloud infrastructure lacks secure and cost-effective mechanisms for discovering and exploiting idle computational capacity.

Thus, despite the potential opportunities, federated cloud computing infrastructure where computational resources are offered by many cooperating participants [101] has seen limited use compared to monolithic offerings from large providers. A knowledge plane can help facilitate federation by adding transparency and accountability to economic transactions in the cloud. These derive from NetQuery's reasoning abstractions, which enable participants to specify application requirements and determine that they hold. When applied to cloud computing, NetQuery reasoning combines network properties, such as those describing enhanced network guarantees from Section 5.1, with properties describing the end host cloud software stacks, such as the allocation of memory, CPU, and I/O bandwidth between different VMs. By enabling buyers to automatically determine whether a cloud provider delivers a necessary level of service, NetQuery reduces the barrier to participation in the cloud marketplace.

[3]This market barrier is orthogonal to other potential sources of cloud inefficiency, such as datacenter network over-subscription, which can strand capacity, and proprietary cloud interfaces, which can cause vendor lock-in.

Reasoning in a federated cloud computing infrastructure should satisfy the following requirements:

- To enable buyers to make sound decisions about the suitability of advertised cloud resources, reasoning should cover the trustworthiness of the underlying information.
- In a federated cloud, intermediaries might buy and resell capacity to hedge or speculate on price changes or to provide value-added services. By making analysis results self-certifying and non-repudiable, the knowledge plane should allow intermediaries to pass analysis results from their original purchase to their customers as supporting evidence for claims about the resold service. Self-certifying and non-repudiable analysis results also serve as an audit trail, which can detect and discourage fraud.
- Commercial infrastructure operators often regard as confidential the internal details of their datacenters and networks. To accommodate such practices, analysis should limit information leakage to acceptable levels.

These requirements closely parallel those of the federated Internet. NetQuery analysis enables applications to check compliance with requirements by establishing characteristics in a remotely verifiable fashion. For instance, when reasoning about the network capacity available to a VM instance, an analysis might examine the network topology, utilization level, and active VM instances, then use NAL to synthesize a proof showing each VM's share of the network. Implementing analysis with sanitizers protects confidential operator information.

5.2.1 Feasibility analysis

Here, we evaluate the feasibility in applying NetQuery to cloud computing by examining the scalability requirements of the knowledge lane, as framed in the context of a *seller* (cloud providers) offering computational resources from a virtualized cloud data center and *buyers* (potential tenants or other providers) seeking to lease computational resources.

The total required service capacity from the knowledge plane is proportional to demand from the seller's devices and abstractions that are represented in it, demand from the buyers' analysis code that retrieve properties from it, and the churn rate (e.g., arrival and departure) of VM instances. As the total number of components and system state update rate increase, the amount of storage and processing power increases. Likewise, buyers induce load as they issue discovery operations to find capacity. To scale to larger datacenter sizes or higher request rates, the knowledge plane can be partitioned across multiple servers, with each server responsible for a subset of tuples.

The following estimates indicate that knowledge planes in typical datacenters can be realized with a small number of NetQuery servers:

Space Let us analyze the knowledge plane requirements for a 20,000 node, non-oversubscribed datacenter. For a network with 1 Gb/s access links in each 48-node rack, 128-port 10 Gb/s aggregation and core switches, partitioned into multiple L2 domains connected by an L3 core[4], this topology requires approximately 450 switches. The tuple representation for each switch consists of the

[4]Datacenter topologies such as [71] which seek to minimize table size and churn will have significantly lower requirements.

ACL filter rules, forwarding table entries, and neighbor list; of these, the ACL rules (100-32,000 entries) and forwarding tables (16,000-32,000 entries) dominate. Given conservative assumptions about the size of ACL and forwarding entries, each switch fits in about 5 MB of space. The tuple representation of each node consists of the hardware description (e.g., memory, CPU, disk) and a list of active VM instances. When dumped in a verbose SNMP format, these host properties fit in under 200 KB. Hence, the total space requirement is 2250 MB (for switches) + 4000 MB (for nodes), which easily fits in memory on a commodity server.

Update rate The state update rate is largely determined by the VM instance churn rate. As VM instances are started and stopped, the node tuples and forwarding tables of every switch in its L2 domain change. Assuming a L2 domain of 500 virtual machines [115] and an average of 4 VM instances per physical node, installing a new VM instance will potentially trigger forwarding table updates in three top-of-rack switches. Under the assumption of 100K VM instance arrivals per day [123], on average 2.3 VM instances enter and exit per second. Thus, the state update rate induced by new instances is minimal.

Query rate Every discovery operation to find capacity involves running an analysis that fetch information from the knowledge plane. Since VMs from a buyer are typically installed after examining bids discovered from several potential sellers, the number of discovery operations is directly proportional to the VM arrival rate. Each analysis in turn can examine tuples from multiple components. Because our cloud resource trading applications only have a small number of unique claims, which are issued and checked in proportion to the number of new customers and VMs, we expect the query rate to be manageable.

5.2.2 Knowledge plane extensions to cloud infrastructure

We now describe how to extend cloud infrastructure to generate trustworthy ground properties even against malicious sellers (e.g., intermediaries or primary suppliers) that have ownership control over the data center infrastructure. We assume the availability of TPMs on every end host in the data center, which is reasonable given that TPMs are already available on commodity server hardware. TPMs bootstrap trust in the end host cloud software stack and fabric controller, thus ensuring that any property they export to the knowledge plane can be tied back to a specific hardware/software platform. As a result, attackers cannot directly spoof properties.

Cloud data centers can be extended to export trustworthy properties by adding shims to intercept information and send it to the knowledge plane. Because cloud datacenters hold similar management state at multiple layers, these shims can be added in several places. The fabric controller maintains a global state store describing how each component is used [1, 149]. Each individual component also export similar state. For example, the fabric controller tracks in its global state store which VM instances are mapped to each compute node in the datacenter, using this data for provisioning and load balancing. At the same time, the hypervisors for each node knows which VM instances are running locally.

Adding shims in the fabric controller's global state store, rather than each datacenter component, has the advantage of providing analyzers with a similar programming model as the fabric controller code, enabling programmers familiar with fabric controllers to apply such experience to writing analyses. Apart from the potential benefits of API reuse, the programming model already defines consistency semantics between the global state and the datacenter components, which DCQuery would inherit. By comparison, using shims at each component to populate DCQuery would introduce another, potentially different, consistency semantics for the knowledge plane's global state.

5.2.3 Protecting the accuracy of network properties

Cloud data centers are potentially vulnerable to attacks that manipulate the execution environment, thereby tricking attested, honest components into reporting false information; such attacks are similar to those in federated networks (Section 3.2).

In virtualized datacenters, virtual switches are widely distributed and can be readily extended with new monitoring mechanisms to prevent and detect such tampering. The virtual switches constantly monitor the network using probes that an attacker cannot distinguish from regular traffic. The statistics from all virtual switches are periodically merged to look for anomalies relative to the purported topology.

For instance, suppose a seller claims to have a non-oversubscribed network. In a non-oversubscribed network, the only bottlenecks can be at the sender's or receiver's access link. To verify this, each virtual switch can be configured to use low priority flows to probe for residual capacity along the path to every destination currently in use by the locally-running VMs. If total capacity (i.e., probe + application traffic) is ever less than the full link capacity, then the fabric controller checks the destination virtual switch to verify that the sender is indeed bottlenecked.

Performing such monitoring in the virtual switch instead of VMs has several advantages. Measurements at the virtual switch are less noisy, since it can see traffic from every local VM. They are also less costly, since a single set of measurements can be amortized across all VMs. Deploying monitoring in virtual switches provides a transition model for legacy cloud data centers, providing anomaly detection in lieu of upgrades to physical switches to perform such detection.

Auditor-generated properties

Some properties that can impact the validity of a claim are not observable by the cloud infrastructure. For instance, the physical security provided by a datacenter depends on who is allowed into the physical facility, and the true redundancy level of a logically-independent set of components is contingent on whether the physical components upon which they depend (e.g., power, cooling, physical fiber links, financial soundness of operator) also have independent failure modes.

Such ground statements come from audits of the business processes and infrastructure. While audits are manual processes conducted by a trusted third party, most datacenters and cloud providers already conduct these audits [73] to support the legal compliance requirements of cloud customers. Thus, the knowledge plane imposes only the small additional cost of publishing the audit results in machine-readable form.

5.2.4 Establishing new guarantees with reference monitors

By monitoring and rewriting control messages, reference monitors [58], implemented in shims that interpose on the interactions between components, provides a lightweight, yet flexible, mechanism for building new guarantees, such as enabling resale of VMs.

Reference monitors can be implemented with few changes to the fabric controller or components. To add a reference monitor, the control interface is modified such that all messages must cross the reference monitor. Reference monitors run in an attested execution environment isolated from the rest of the system, such as a small VM.

5.2.5 Applications

Reselling VMs

A VM owner that wants to enable reselling of that VM permanently gives its access credentials to that VM to a reference monitor; after this, only the reference monitor can directly manipulate the resource. The reference monitor then issues a credential to the resource owner that allows it to access the resource through the reference monitor. The reference monitor is well-known to potential buyers and is trusted to implement delegation; its proper installation can be remotely verified with attestation. When the resource owner resells the VM to a buyer, the reference monitor revokes all access from the owner and issues a new credential to the buyer. To protect the data confidentiality of the buyer, the reference monitor scrubs all state from the VM before returning control to the owner.

Transparent cloud

A knowledge plane brings transparency to cloud resource discovery and provisioning, eliminating the need for costly trust establishment. By removing this market barrier, transparency improves pricing efficiency and resource utilization. Given an offer of computational resources from a seller, a buyer can use the knowledge plane to verify the seller's claims. For instance, a buyer can verify a set of performance claims by analyzing the network topology and number of active VMs to determine maximum network capacity, oversubscription level, and redundancy.

Many types of claims are much easier for buyers to check with a knowledge plane than with probing. Cloud sellers might offer premium units of computation that are indistinguishable in function and performance from regular computation yet have significantly different isolation and provisioning properties. For instance, tenants may pay extra for VMs that are physically segregated from other tenants, minimizing potential for information leakage [12]. Tenants may also pay to reserve physical hardware for future use [14]. In both cases, analysis combined with triggers and reference monitors can verify that these promises are kept.

Once buyers can readily verify seller claims, cloud participants have considerably more flexibility in mitigating the risks of operating large-scale applications. To hedge against long-term capacity planning risks, cloud providers can sell prepaid allotments of capacity (e.g., reserved instances or long-term contracts); these shift capacity planning risk to the customer. Transparency enables buyers of these reserved allotments to resell entire contracts or sub-lease idle capacity. Similarly, many corporations install large private clouds; they can recover part of the cost of these clouds by partitioning off idle parts of the cloud and offering it for public use; this capacity can later be safely reclaimed with a TPM-attested reboot. This approach provides better control over isolation and hardware reuse than a hosted private cloud [12].

Federated cloud commodities exchange

Trading computational capacity on commodities exchanges can improve pricing efficiency and streamline discovery and exploitation of idle resources [101]. Commodities are typically categorized into equivalent quality bins to abstract away unimportant details and to maximize fungibility and liquidity. The bilateral transparent cloud naturally generalizes into a cloud commodities exchange.

The exchange defines a commodities schema for buyer bids and seller offers of computational resources. Because buyers and sellers can vary in how much they know about their performance requirements and achievable SLA, these schemas should cover different levels of precision. Because many cloud applications consist of multiple closely-coupled instances, these schemas should extend to cover multiple node bids and offers. For instance, a buyer that wants to deploy a new service may not yet know how much network capacity is needed; such buyers would specify an estimated set of instance types and request duration, with network capacity left as best-effort. A buyer that needs a high performance map-reduce cluster might issue a bid for a set of high-CPU instances connected over a full bisection bandwidth network. [5]

[5]A full bisection bandwidth network of size N is one where, for all bisections (i.e., partition into two sets of size $N/2$ each), the aggregate bandwidth between the sets is $N/2$ times the capacity of a single host uplink. That is, all nodes can simultaneously communicate at full speed.

Applications can benefit from still more expressive schemas. Schemas that embed location and WAN capacity information enable applications such as content distribution that are performance-sensitive. Claims in all preceding schemas are straightforward to check with analyses.

Allowing both buyers and sellers to post price information yields a double auction market, which generally leads to more efficient prices [65]. Posting buyer requirements is often more straightforward to implement, since the buyer knows what kind of datacenter would be appropriate. By comparison, a seller can offer many possible VM instance types from a single pool of resources. There is likely a tradeoff between pricing efficiency, the diversity of offers posted for this pool, and overhead of failed buyer/seller matches caused by stale offers. For instance, a seller may simultaneously offer many instances but not have enough resources to accept bids for all offers at once. If multiple bids arrive at the same time, some will not succeed and need to be routed to another seller.

Much as in existing financial exchanges, the cloud commodities exchange enables intermediaries to compose multiple offers into yet more complex transactions. For instance, intermediaries can engage in market making or arbitrage, such as sub-dividing one instance type into another, or merge multiple commodities markets into one. Having a knowledge plane enables each of these while preserving transparency.

5.2.6 Open problems

Deployment incentives

Many cloud participants would benefit from a federated cloud. Transparency benefits users by stimulating competition and gives users greater flexibility in balancing costs and risk. It also facilitates innovation in cloud intermediaries.

Smaller cloud providers and private cloud operators are incentivized to deploy a knowledge plane: with the baseline knowledge plane deployment, they can increase their value relative to the incumbent public cloud providers. They also benefit from installing other value-adding upgrades, such as adding support for nested hypervisors to facilitate arbitrary repackaging and better control over scheduling [28]. These benefits do not have critical mass of provider support as a prerequisite, but rather could start paying off as soon as a knowledge plane is supported in commodity cloud stacks and middleware.

By comparison, the deployment incentives are uncertain for the major incumbent cloud providers since they benefit from existing market barriers. Allowing customers to resell capacity can change the load statistics, since harvesting arbitrage opportunities and spare cycles would be a zero-sum game with the customers. On the other hand, allowing customers to re-transfer risk may result in more up-front sales of reserved capacity, and a more efficient market may provide better demand prediction signals. The structure of the cloud market may also have too many providers for the major incumbents to collude. Resolving this tension is an open economics and business question.

CHAPTER 6

SEAWALL DESIGN AND IMPLEMENTATION

Data centers that provide fine-grained control over network capacity allocation are mutually beneficial for data center operator and tenants. The diverse mix of tasks and VMs in a typical cloud data center workload can lead to performance interference and denial-of-service attacks; such uncertainty raises tenant concerns about moving critical applications to the cloud. Operators can leverage better control over network capacity allocation to offer tiers of service, allowing tenants that have more stringent performance requirements to pay more for a guaranteed level of service.

By satisfying the following requirements, Seawall can be applied across diverse range of data center networks:

- **Require no changes to network topology or hardware.** Recently, many data center network topologies have been proposed [71, 10, 77, 90]. Cost-benefit trade-offs indicate that topologies should be matched to intended usage. For example, EC2 supports two network topologies targeting different applications. Network-bottlenecked high performance computing (HPC) applications can use VMs connected to a full bisection bandwidth network. Applications without such stringent network performance requirements can use less expensive VMs that are connected to a conventional oversubscribed tree topology. To be widely applicable, mechanisms to share the network should be agnostic to network topology.
- **Scale to large numbers of tenants and high churn.** Any network sharing mechanism would need to scale to support the workloads seen in large scale data centers, such as those used for cloud computing.

- **Enforce sharing without sacrificing efficiency.** Statically apportioning fractions of the bandwidth improves sharing at the cost of efficiency and can result in bandwidth fragmentation that makes it harder to accommodate new tenants. At the same time, a tenant with pent up demand can use no more than its reservation even if the network is idle. Poor network utilization degrades performance and can strand computational resources, resulting in increased costs for the data center operator.

Seawall has few dependencies on the physical network, since it relies on neither special switch functionality, such as per-flow state or rate limiting, nor assumptions about network topology. Instead, Seawall relies on *congestion-controlled tunnels* implemented in hosts. Seawall does benefit from measurements at switches, if they are available. Seawall scales to large numbers of tenants and handles high churn because physical switches do not need to be reconfigured as tenants, VMs, or tasks come and go, reducing processing demands on switch control processors.

Seawall's congestion control architecture and algorithms differ from those of existing end-to-end approaches. TCP is the de facto standard for sharing network capacity. While it provides end-to-end congestion control, it does not satisfy our requirements. In principle, TCP allocates bottleneck capacity in a flow-oriented fashion, with concurrent flows equally sharing the bandwidth capacity of bottleneck links. However, malicious or selfish tenants can easily bypass TCP by changing their TCP implementation. Even if TCP were mandatory, the flow-oriented policy provides neither flexible nor meaningful control over capacity allocation. Many operators today deploy differentiated tiers of service that allows customers to pay more for faster VMs. Service tiers are currently differentiated by allocating of local resources, such as CPU, memory, and storage, in proportion to VM price [13]. But flow-oriented allocations do not support such proportional allocation. For instance, when two tenants with the same aggregate demand but differing number of flows share a data center, the tenant with more flows will receive more network capacity. Indeed, hosts can arbitrarily increase their share of the network simply by opening more flows.

Seawall's packet processing architecture and control loop addresses these concerns. Conformance with Seawall control is mandatory: processing occurs in the cloud computing stack (e.g., virtual switch or platform network stack), where it is isolated from tenant code and cannot be bypassed. Seawall's control loop is link-oriented, equally allocating the bandwidth of bottleneck links between the nodes using those links

To illustrate the broad applicability of Seawall, this chapter describes a Seawall architecture and implementation that depends only on functionality found in existing virtual switches and network stacks. This design provides operators and tenants with all of the benefits of improved control over network bandwidth allocation.

When combined with trusted packet processors and NetQuery, Seawall's guarantees can be remotely verified by other participants in a federated cloud computing infrastructure. Adding TPMs to end hosts helps to protect Seawall from attack: since Seawall uses a distributed control loop, nodes that do not conform to the protocol can skew the resulting bandwidth allocations. TPMs can detect nodes with non-conformant software stacks, thereby preserving the control loop guarantees.

6.1 Empirical motivation

To understand the limitations of existing network allocation schemes, we examine two types of clusters that consist of several thousands of servers and are used in production. The first type is that of public infrastructure cloud services that rent virtual machines along with other shared services such as storage and load balancers. In these datacenters, clients can submit arbitrary VM images and choose which applications to run, who to talk to, how much traffic to send, when to send that traffic, and what protocols to use to exchange that traffic (TCP, UDP, # of flows). The second type is that of platform cloud services that support map-reduce workloads. Consider a map-reduce cluster that supports a search engine. It is used to analyze logs and improve query and advertisement relevance. Though this cluster is shared across many users and business groups, the execution platform (i.e., the job compiler and runtime) is proprietary code controlled by the datacenter provider.

Through case studies on these datacenters we observe how the network is shared today, the problems that arise from such sharing, and the scalability requirements for an improved sharing mechanism.

In all examined data centers, the servers have multiple cores, multiple disks, and tens of GBs of RAM. The network is a tree like topology [10] with 20–40 servers in a rack and a small over-subscription factor on the upstream links of the racks.

6.1.1 Performance interference in infrastructure cloud services

Recent measurements demonstrate considerable variation in network performance metrics – medium instances in EC2 experience throughput that can vary by 66% [102, 152]. We conjecture, based on anecdotal evidence, that a primary reason for the variation is the inability to control the network traffic share of a VM.

Unlike CPU and memory, network usage is harder to control because it is a distributed resource. For example, consider a simple strawman where each VM's network share is statically limited to a portion of the host's NIC rate (the equivalent of assigning the VM a fixed number of cores or a static memory size). A tenant with many VMs can cumulatively send enough traffic to overflow the receiver, some network link en route to that host, or other network bottlenecks. Some recent work [128] shows how to co-locate a malicious VM with a target VM. Using this, a malicious tenant can degrade the network performance of targeted victims. Finally, a selfish client, by using variable numbers of flows, or higher rate UDP flows, can monopolize network bandwidth.

We note that out-of-band mechanisms to mitigate these problems exist. Commercial cloud providers employ a combination of such mechanisms. First, the provider can account for the network usage of tenants (and VMs) and quarantine or ban the misbehavers. Second, cloud providers might hide details regarding network topology and VM placement to make it harder for a malicious client to co-locate with target VMs. However, neither approach is fool-proof. Selfish or malicious traffic can mimic legitimate traffic, making it hard to distinguish. Further, obfuscation schemes may not stop a determined adversary.

6.1.2 Performance interference in data intensive workloads

Data intensive workloads, such as map-reduce, use the network to shuffle substantial volumes of data and intermediate results between nodes within a cluster. Network overheads can have significant impact on end-to-end performance metrics. To evaluate the impact of network performance interference for such workloads, we obtained detailed logs over several days from Cosmos [37], a production cluster with thousands of servers that supports the Bing search engine. The logs document the begin and end times of jobs, tasks and flows.

This cluster exhibits many instances of high network load during which performance interference is apparent. A few entities (jobs, background services) contribute a substantial share of the traffic [91]. Tasks that move data over congested links suffer collateral damage – they are more likely to experience failures and become stragglers at the job level [16, 91].

Uniquely, however, we find that the de facto way of sharing the network leads to poor schedules. This is because schedulers for map-reduce platforms [87, 157] explicitly allocate local resources such as compute slots and memory. But, the underlying network primitives prevent them from exerting control over how tasks share the network. Map-reduce tasks naturally vary in the number of flows and the volume of data moved – a map task may have to read from just one location but a reduce task has to read data from all the map tasks in the preceding stage. Figure 6.1 shows that of the tasks that read data across racks, 20% of the tasks use just one flow, another 70% of the tasks vary between 30 and 100 flows, and 2% of the tasks use more than 150 flows. Figure 6.2 shows that this variation is due to the role of the task.

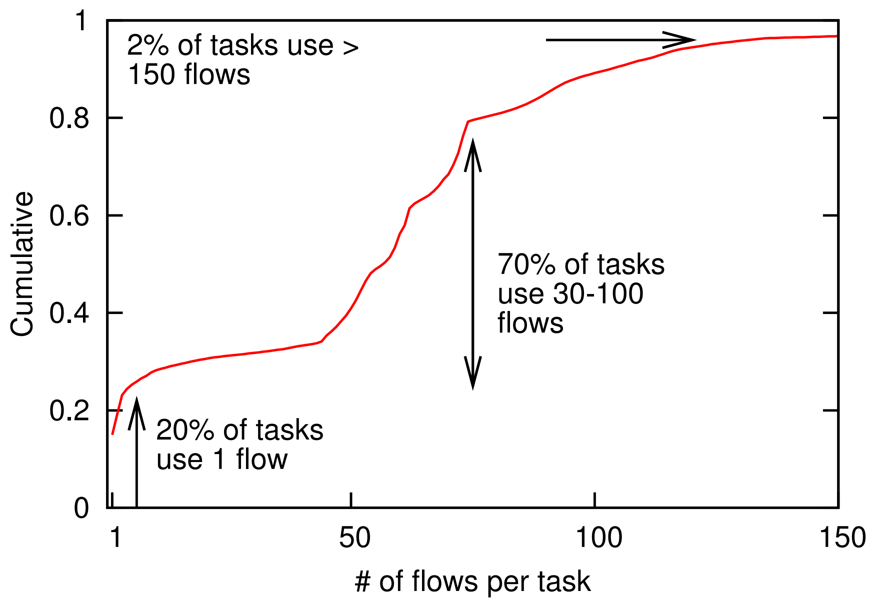
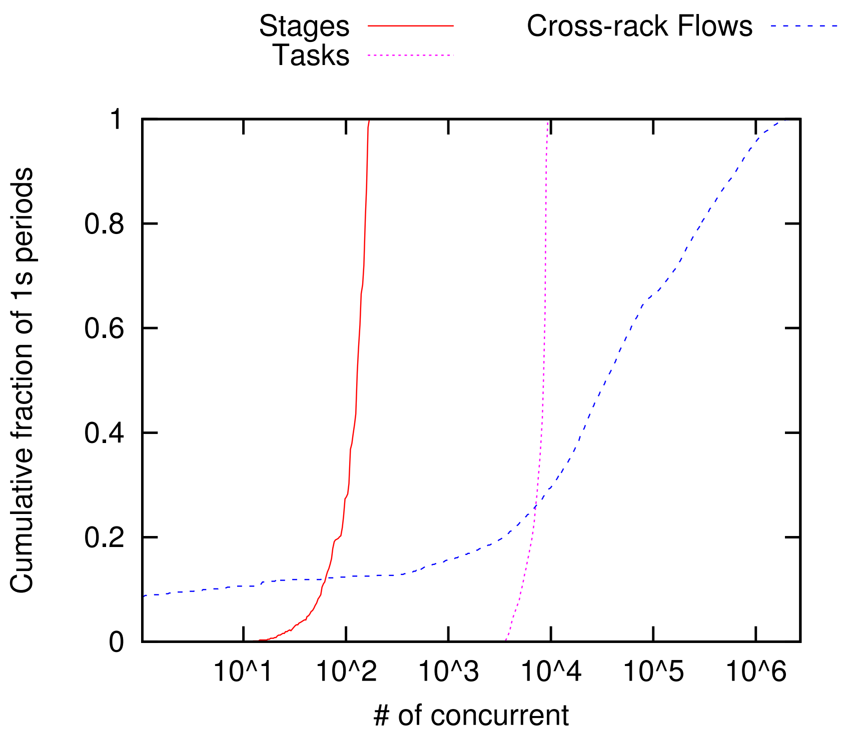


Figure 6.1: Distribution of the number of flows per task in Cosmos.

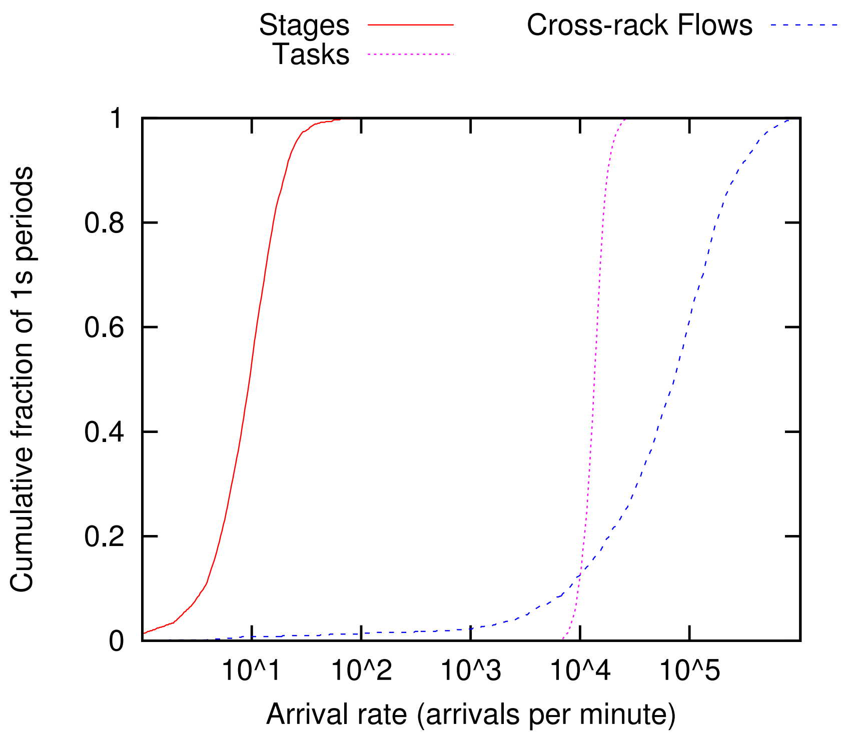
Task type	**# flows per task**	**% of net tasks**
Aggregate	56.1	94.9
Partition	1.2	3.7
Extract	8.8	.2
Combine	2.3	1.0
Other	1.0	.2

Figure 6.2: Variation in number of flows per task is due to the role of the task

Because reduce tasks use a large number of flows, they starve other tasks that share the same paths. Even if the scheduler is tuned to assign a large number of compute slots for map tasks, a small number of reduce tasks will cause these map tasks to be bottlenecked on the network. Thus, the compute slots held by the maps make little progress.



(a) Cosmos: Scale



(b) Cosmos: Churn

Figure 6.3: Scale and churn seen in the observed datacenter.

6.1.3 Magnitude of scale and churn

The suitability of a network allocation scheme is in part determined by the workload that it manages; a suitable scheme should be able to express policies with sufficient control and granularity while scaling to the number of entities under management. In the Cosmos cluster, the number of traffic classes to share bandwidth among is large and varies frequently. Figure 6.3(a) shows the distribution of the number of concurrent entities that share the examined Cosmos cluster. We see that at median, there are 500 stages (e.g., map, reduce, join), 10^4 tasks and 10^5 flows in the cluster. The number of traffic classes required is at least two orders of magnitude larger than is feasible with current CoS tags or the number of WFQ/DRR queues that switches can handle per port.

Figure 6.3(b) shows the distribution of the number of new arrivals in the observed cluster. Note that the x-axis is again in log scale. At median, 10 new stages, 10^4 new tasks and $5 * 10^4$ new flows arrive in the cluster every minute. Anecdotal analysis of EC2, based on decoding the instance identifiers, concluded that O(10^4) new VM instances are requested each day [130]. Updating VLANs or re-configuring switches whenever a VM arrives is several orders of magnitude more frequent than is achievable in today's enterprise networks.

Each of the observed data centers is large, with up to tens of thousands of servers, thousands of ToR switches, several tens of aggregation switches, load balancers, etc. Predicting traffic is easier in platform datacenters, such as Cosmos, wherein high level descriptions of the jobs are available. However, the scale and churn numbers indicate that obtaining up-to-date information (for instance, at minute granularity) may be a practical challenge. In cloud datacenters like EC2 traffic is even harder to predict because customer traffic is unconstrained.

6.2 Design

Seawall exposes control over network allocation through the *network weight* abstraction. A network weight is associated with each entity that is sharing the network. The entity can be any traffic source that is confined to a single node, such as a VM, process, or collection of port numbers, but not a tenant or set of VMs. On each link in the network, Seawall provides the entity with a bandwidth share that is proportional to its weight; i.e., an entity k with weight w_k sending traffic over link l obtains this share of the total capacity of that link $Share(k, l) = \frac{w_k}{\Sigma_{i \in Active(l)} w_i}$. Here, $Active(l)$ is the set of entities actively sending traffic across l. The allocation is end-to-end, i.e., traffic to a destination will be limited by the smallest $Share(k, l)$ over links on the path to that destination. The allocation is also work-conserving: bandwidth that is unused because the entity needs less than its share or because its traffic is bottlenecked elsewhere is re-apportioned among other users of the link in proportion to their weights.

Assigning the same weight to all entities divides bandwidth in a max-min fair fashion. By specifying equal weights to VMs, a public cloud provider can avoid performance interference from misbehaving or selfish VMs (Section 6.1.1). The control loop also accommodates dynamic adjustments to weights, reconverging rapidly after such changes. We explore examples of configuring weights and enforcing global allocations in Section 6.2.6.

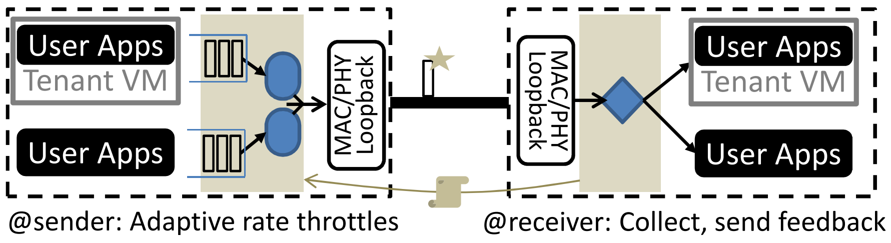


Figure 6.4: **Seawall's division of functionality.** New components are shaded gray.

6.2.1 Data path

To achieve the desired sharing of the network, Seawall sends traffic through congestion-controlled tunnels. As shown in Figure 6.4, these tunnels are implemented within a shim layer that intercepts all packets entering and leaving the server. At the sender, each tunnel is associated with an allowed rate for traffic on that tunnel, implemented as a rate limiter. The receive end of the tunnel monitors traffic and sends congestion feedback back to the sender. A bandwidth allocator corresponding to each entity uses feedback from all of the entity's tunnels to adapt the allowed rate on each tunnel. The bandwidth allocators take the network weights as parameters, work independently of each other, and together ensure that the network allocations converge to their desired values.

The Seawall shim layer is deployed to all servers in the data center by the fabric controller that is responsible for provisioning and monitoring these servers. To ensure that only traffic controlled by Seawall enters the network, a provider can use attestation-based 802.1x authentication to disallow servers without the shim from connecting to the network.

The feedback to the control loop is returned at regular intervals, spaced T apart. It includes both explicit control signals from the receivers as well as congestion feedback about the path. Using the former, a receiver can explicitly

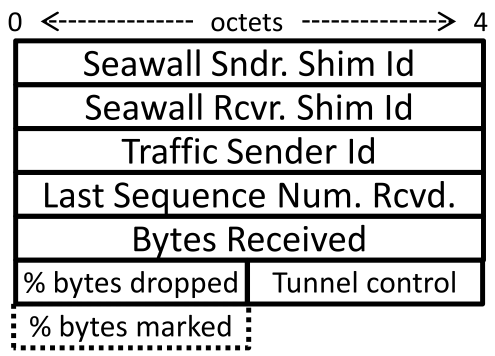


Figure 6.5: **Content of Seawall's feedback packet.**

block or rate-limit unwanted traffic. Using the latter, the bandwidth allocators adapt allowed rate on the tunnels. To help the receiver prepare congestion feedback, the shim at the sender maintains a per-tunnel (i.e., per (sending entity, destination) pair) byte sequence number. The sender shim stamps outgoing packets with the corresponding tunnel's current sequence number. The receiver detects losses in the same way as TCP, by looking for gaps in the received sequence number space. At the end of an interval, the receiver issues feedback that reports the number of bytes received and the percentage of bytes deemed to be lost (Figure 6.5). Optionally, if ECN is enabled along the network path, the feedback also relays the fraction of packets received with congestion marks.

```
1:  .Begin (weight $W$)
2:  { rate $r \leftarrow I$, weight $w \leftarrow W$ }            ▷ Initialize
3:  .TakeFeedback (feedback $f$, proportion $p$)
4:  {
5:  if feedback $f$ indicates loss then
6:      $r \leftarrow r - r * \alpha * p$                ▷ Multiplicative Decrease
7:  else
8:      $r \leftarrow r + w * p$                    ▷ Weighted Additive Increase
9:  end if
10: }
```

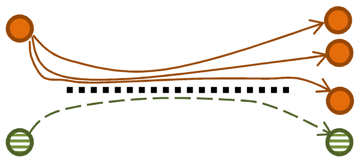
Figure 6.6: **A strawman bandwidth allocator.** An instance of this allocator is associated with each (entity, tunnel) pair.

6.2.2 Strawman

Consider the strawman bandwidth allocator in Figure 6.6. Recall that the goal of the bandwidth allocator is to control the entity's network allocation as per the entity's network weight. It works as follows: when feedback indicates loss, it multiplicatively decreases the allowed rate by α. Otherwise, the rate increases by an additive constant.

This simple strawman satisfies some of our requirements. By making the additive increase step size a function of the entity's weight, the equilibrium rate achieved by an entity will be proportional to its weight. Unused shares are allocated to tunnels that have unsatisfied demand, favoring efficiency over strict reservations. The control loop is distributed, requiring no global coordination since all nodes interact indirectly through congestion signals and the convergence properties of the control loop. Further, when weights change, rates re-converge quickly.

This allocator is a variant of weighted additive increase, multiplicative decrease (AIMD); any other flow-oriented distributed control loop [11, 6, 117, 127] can be adapted similarly to achieve the same properties, so long as it can extend to provide weighted allocations (e.g., see control loop transformations from MulTCP or MPAT [49, 138]). Distributed control loops are sensitive to variation in RTT. Seawall avoids this by using a constant feedback period T, chosen to be larger than the largest RTT of the intra-datacenter paths controlled by Seawall. Conservatively, Seawall considers no feedback within a period of T as if a feedback indicating loss was received.



Allocation Strategy	Share of Bottleneck for orange entity
Pair-wise	$\propto$ to # of destinations
Seawall	0.5

Figure 6.7: **Limitations of flow-oriented bandwidth allocation.** When entities talk to different numbers of destinations, flow-oriented allocation of bandwidth is not sufficient. Reduce tasks behave like the orange entity while maps resemble the green.

Simply applying AIMD, or related (i.e., TCP-friendly) control loops, on a per-tunnel basis does not Seawall's target level of control over bandwidth allocation. Suppose a tenant has N VMs and opens flows between every pair of VMs. This results in a tunnel between each VM; with one AIMD loop per tunnel, the tenant can achieve an allocation of the bottleneck link that varies quadratically with its size in VMs. Thus, larger tenants can overwhelm smaller tenants (Figure 6.7).

Seawall improves on the mechanism in Figure 6.6 in three ways. First, it has a unique technique to combine feedback from multiple destinations. By doing so, an entity's share of the network is governed by its network weight and is independent of the number of tunnels it uses (Section 6.2.3). The resulting policy is consistent with how cloud providers allocate other resources, such as compute and memory, to a tenant, yet is a significant departure from prior approaches to network scheduling with distributed control protocols. Second, the sawtooth behavior of AIMD leads to poor convergence on paths with high bandwidth-delay product. To mitigate this, Seawall modifies the adaptation logic to converge quickly and stay at equilibrium longer (Section 6.2.4). Third, mixing traffic with different levels of responsiveness to congestion signals (e.g., TCP vs. UDP) within Seawall can alter the bandwidth allocations achieved by Seawall; Seawall addresses this with network stack modifications (Section 6.2.5).

6.2.3 Bandwidth Allocator

Each sending entity is associated with a separate instance of the bandwidth allocator. The bandwidth allocator takes as input the network weight of that entity and the congestion feedback from all the receivers that the entity is communicating with, then generates the allowed rate on each of the entity's tunnels. It has two parts: a distributed congestion control loop that computes the entity's cumulative share on each link and a local scheduler that divides that share among the various tunnels.

Step 1: Use distributed control loops to determine per-link, per-entity share. The ideal feedback for the Seawall control loop would be per-link. It would include the cumulative usage of the entity across all the tunnels on this link, the total load on the link, and the network weights of all the entities using that link. Such feedback is possible if switches implement explicit feedback (e.g., XCP, QCN) or from programmable switch sampling (Section 5.1.3). Lacking these, the baseline Seawall relies only on existing congestion signals such as end-to-end losses or ECN marks. These signals identify congested paths, rather than links.

To approximate link-level congestion information using path-level congestion signals, Seawall uses a heuristic based on the observation that a congested link causes losses in many tunnels using that link. The logic is described in Figure 6.8. One instance of this allocator is associated with each entity and maintains separate per-link instances of the distributed control loop (rc_l). Assume for now that rc is implemented as per the strawman from Figure 6.6, though we will replace it with the mechanism in Figure 6.9. The sender shim stores the feedback from each destination, and once every period T, applies all the feedback cumulatively (lines 8–10). The heuristic scales the impact of feedback from a given destination in proportion to the volume of traffic sent to that destination by the shim in the last period (line 7, 10).

To understand how this helps, consider the example in Figure 6.7. An instance of Figure 6.8, corresponding to the orange entity, cumulatively applies the feedback from all three destinations accessed via the bottleneck link to the single distributed control loop object representing that link. Since the proportions sum to one across all destinations, the share of the orange entity will increase by only as much as that of the green entity.

Rather than invoking the distributed control loop once per destination, Figure 6.8 computes just three numbers per link – the proportions of total feedback indicating loss, ECN marks, and loss-free delivery, and invokes the distributed control loop once with each.

```
1:  .Begin (weight $W$)
2:  { $rc_l$.Begin($W$) $\forall$ links $l$ used by sender }          ▹ Initialize
3:  .TakeFeedback (feedback $f_{dest}$)
4:  { store feedback }
5:  .Periodically ()
6:  {
7:  proportion of traffic to $d$, $p_d = \frac{f_d.bytesRcvd}{\sum f_i.bytesRcvd}$
8:  for all destinations $d$ do
9:      for all links $l \in PathTo(d)$ do
10:         $rc_l$.TakeFeedback($f_d$, $p_d$)
11:     end for
12: end for
13:                          ▹ $rc_l$ now contains per-link share for this entity
14: $n_l \leftarrow$ count of dest with paths through link $l$
15:                                              ▹ $r_d$ is allowed rate to $d$
16: $r_d \leftarrow \min_{l \in PathTo(d)} \left( \left( \beta p_d + \frac{1-\beta}{n_l} \right) rc_l.rate \right)$
17: }
```

Figure 6.8: **Seawall's bandwidth allocator**. A separate instance of this algorithm is associated with each entity. It combines per-link distributed control loops (invoked in lines 2, 10) with a local scheduler (line 16).

Step 2: Convert per-link, per-entity shares to per-link, per-tunnel shares. Next, Seawall runs a local allocator to assign rate limits to each tunnel that respects the entity's per-link rate constraints. Dividing each link's allowed rate evenly across all downstream destinations leads to wasted bandwidth if the demands across destinations vary. For the example in Fig. 6.7, an even allocation leads to a $\frac{1}{3}'rd$ share of the bottleneck link to the three destinations of the orange entity. If the orange entity has demands $(2x, x, x)$ to the three destinations and the bottleneck's share for this entity is $4x$, dividing evenly causes the first destination to get no more than $\frac{4x}{3}$ while bandwidth goes wasted. Hence, Seawall apportions link bandwidth to destinations as shown in line 16, Figure 6.8. The intuition is to adapt the allocations to match the demands. Seawall uses an exponential moving average that allocates β fraction of the link bandwidth proportional to current usage and the rest evenly across destinations. By default, we use $\beta = .9$. Revisiting the $(2x, x, x)$ example, note that while the first destination uses up all of its allowed share, the other two destinations do not, causing the first to get a larger share in the next period. In fact, the allowed share of the first destination converges to within 20% of its demand in four iterations.

Finally, Seawall converts these per-link, per-destination rate limits to a tunnel (i.e., per-path) rate limit by computing the minimum of the allowed rate on each link on the path. Note that Figure 6.8 converges to a lower bound on the per-link allowed rate. At bottleneck links, this bound is tight. At other links, such as those used by the green flow in Figure 6.7 that are not the bottleneck, Figure 6.8 can under-estimate their usable rate. This is harmless because all paths from green are already bottlenecked elsewhere and cannot push more traffic through such links. When the green entity begins using these links with under-estimated capacity with new paths that are not bottlenecked, the estimated capacity will increase.

```
.Begin (weight $W$)
{ rate $r \leftarrow I$, weight $w \leftarrow W$, $c \leftarrow 0$, $inc \leftarrow 0$ }                ▷ Init
.TakeFeedback (feedback $f$, proportion $p$)
{
$c \leftarrow c + \gamma * p * (f.bytesMarked - c)$
                                        ▷ maintain smoothed estimate of congestion
if $f.bytesMarked > 0$ then
    $r_{new} \leftarrow r - r * \alpha * p * c$                  ▷ Smoothed mult. decrease
    $inc \leftarrow 0$
    $t_{lastdrop} \leftarrow now$
    $r_{goal} \leftarrow (r > r_{goal})?r : \frac{r+r_{new}}{2}$
else                                                             ▷ Increase rate
    if $r < r_{goal}$ then                    ▷ Less than goal, concave increase
        $\Delta t = \min\left(\frac{now-t_{lastdrop}}{T_s}, .9\right)$
        $\Delta r = \delta * (r_{goal} - r) * (1 - \Delta t)^3$
        $r \leftarrow r + w * p * \Delta r$
    else                                      ▷ Above goal, convex increase
        $r \leftarrow r + p * inc$
        $inc \leftarrow inc + w * p$
    end if
end if
}
```

Figure 6.9: **Seawall's distributed control loop.** An instance of this allocator is associated with each (link, entity) pair. Note that Figure 6.8 invokes this loop (lines 3, 10).

6.2.4 Improving the Rate Adaptation Logic

Weighted AIMD suffers from inefficiencies as adaptation periods increase, especially for paths with high bandwidth-delay product [93], such as those found in datacenters. Seawall uses control laws from CUBIC [127] to achieve faster convergence, longer dwell time at the equilibrium point, and higher utilization than AIMD. As with weighted AIMD, Seawall modifies the control laws to support weights and to incorporate feedback from multiple destinations. If switches support ECN, Seawall also incorporates the control laws from DCTCP [11] to further smooth out the sawtooth and reduce queue utilization at the bottleneck, resulting in reduced latency, less packet loss, and improved resistance against incast collapse.

The resulting control loop is shown in Figure 6.9; the stability follows from that of CUBIC and DCTCP. Though we describe a rate-based variant, the equivalent window based versions are feasible and we defer those to future work. We elaborate on parameter choices in Section 6.2.6. Lines 14-17 cause the rate to increase along a concave curve, i.e., quickly initially and then slower as rate nears r_{goal}. After that, lines 18-19 implement convex increase to rapidly probe for a new rate. Line 5 maintains a smoothed estimate of congestion, allowing multiplicative decreases to be modulated accordingly (line 8) so that the average queue size at the bottleneck stays small.

6.2.5 Nesting Traffic Within Seawall

Nesting traffic of different types within Seawall's congestion-controlled tunnels can result in corner-case interactions between Seawall's control loop. If a sender always sends less than the rate allowed by Seawall, it may never see any loss causing its Seawall-allowed rate to increase to infinity. This can happen if the sender's flows are low rate (e.g., web traffic) or are limited by send or receive windows (flow control). Such a sender can launch a short overwhelming burst of traffic. Hence, Seawall clamps the rate allowed to a sender to a multiple of the largest rate the sender has used in the recent past. Clamping rates is common in many control loops, such as XCP [93], for similar reasons. The specific choice of clamp value does not matter as long as it is larger than the largest possible bandwidth increase during a Seawall change period.

UDP and TCP flows behave differently under Seawall. While a full burst UDP flow immediately uses all the rate that a Seawall tunnel allows, a set of TCP flows can take several RTTs to ramp up; the more flows, the faster the ramp-up. Since slower ramp up usually results in lower shares, UDP will generally receive larger allocations. Hence, Seawall modifies the network stack to defer congestion control to Seawall's shim layer. All other TCP functionality, such as flow control, loss recovery, and in order delivery remain as before.

The mechanics of re-factoring are similar to Congestion Manager (CM) [26]. Each TCP flow queries the appropriate rate limiter in the shim (e.g., using shared memory) to see whether a send is allowed. Flows that have a backlog of packets register callbacks with the shim to be notified when they can next send a packet. In virtualized settings, the TCP stack defers congestion control to the shim by expanding the paravirtualized NIC interface. Even for tenants that bring their own OSes, the performance gain from refactoring the stack incentivizes adoption.

6.2.6 Discussion

This section discusses details deferred from the preceding description of Seawall.

Handling WAN traffic Traffic entering and leaving the datacenter is subject to more stringent DoS scrubbing at pre-defined chokepoints and, because WAN bandwidth is a scarce resource, is carefully rate-limited and metered. We do not expect Seawall to be used for such traffic. However, if required, edge elements in the datacenter, such as load balancers or gateways, can funnel all incoming and outgoing traffic into Seawall tunnels implemented in those edge elements.

Mapping paths to links To run Seawall, each sender requires path-to-link mapping for the paths that it is sending traffic on (line 10, Figure 6.8). A sender can acquire this information independently, for example via a few traceroutes. In practice, however, this is much easier. Since the fabric controller manages the data center network, it is aware of the topology and already has a signalling path to end hosts for configuration updates. Topology changes (e.g., due to failures and reconfiguration) are rare and can be disseminated automatically by these systems. Indeed, many pieces of today's datacenter ecosystem already use topology information (e.g., Map-Reduce schedulers [87] and VM placement algorithms) to optimize workload placement.

Choosing network weights Seawall provides several ways to define the sending entity and the corresponding network weight. The precise choices depend on the datacenter type and application. When VMs are spun up in a cloud datacenter, the fabric sets the network weight of that VM alongside weights for CPU and memory. The fabric can change the VMs weight, if necessary, and Seawall re-converges rapidly. However, a VM cannot change its own weight. The administrator of a cloud datacenter can assign equal weights to all VMs, thereby avoiding performance interference, or assign weights in proportion to the size or price of the VM.

In contrast, the administrator of a platform datacenter can empower trusted applications to adjust their weights at run-time (e.g., with socket options). Here, Seawall can also be used to specify weights at other granularities, such as per executable (e.g., background block replicator), per process, or per port range. The choice of weights could be based on information maintained by cluster schedulers. For example, a map-reduce scheduler can assign the weight of each sender feeding a task in inverse proportion to the aggregation fan-in of that task, which the scheduler knows at all execution stages. This ensures that each task obtains the same bandwidth (Section 6.1.2). Similarly, the scheduler can boost the weight of outlier tasks that are starved or are blocking many other tasks [16], thereby improving job completion times.

Enforcing global allocations Seawall has so far focused on enforcing the network share of a local entity (VM, task etc.). This is complementary to prior work on Distributed Rate Limiters (DRL) [124] which controls the aggregate rate achieved by a collection of entities. Controlling just the aggregate rate is vulnerable to DoS attacks: a tenant might focus the traffic of all of its VMs on a shared service (such as storage) or link (e.g., ToR containing victim tenant's servers), thereby interfering with the performance of other tenants while remaining under its global bandwidth cap. Running Seawall alongside DRL, with the end host shim rate limiting entities to the minimum of the rate allowed by Seawall and the rate allowed by DRL will protect the network against such attacks.

Choosing parameters Whenever we adapt past work, we follow their guidance for parameters. Of the parameters unique to Seawall, their specific values have the following impact. Reducing the feedback period T makes Seawall's adaptation logic more responsive at the cost of more overhead. We recommend choosing $T \in [10, 50]\ ms$. The multiplicative factor α controls the decrease rate. With the CUBIC/DCTCP control loop (see Figure 6.9), Seawall is less sensitive to α than the AIMD control loop, since the former ramps back up more aggressively. In Figure 6.8, β controls how much link rate is apportioned evenly versus based on current usage. With a larger β, the control loop reacts more quickly

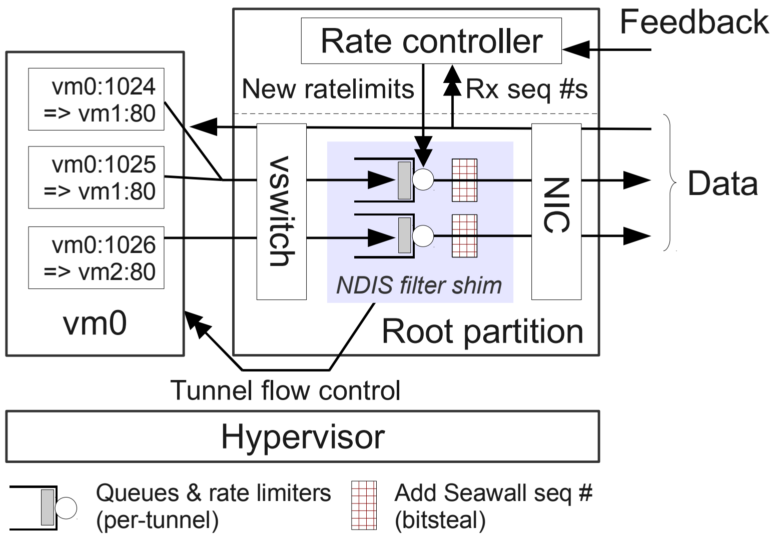


Figure 6.10: The Seawall prototype is split into an in-kernel NDIS filter shim (shaded gray), which implements the rate limiting datapath, and a userspace rate adapter, which implements the control loop. Configuration shown is for infrastructure data centers.

to changing demands but delays apportioning unused rate to destinations that need it. We recommend $\beta > .8$.

6.3 Prototype

The shim layer of our prototype is built as an NDIS packet filter (Figure 6.10). It interposes new code between the TCP/IP stack and the NIC driver. In virtualized settings, the shim augments the vswitch in the root partition. Our prototype is compatible with deployments that use the Windows 7 kernel as the server OS or as the root partition of Hyper-V. The shim can be adapted to other OSes and virtualization technologies, e.g., to support Linux and Xen, one can reimplement it as a Linux network queuing discipline module. For ease of experimentation, the logic to adapt rates is built in user space whereas the filters on the send side and the packet processing on the receive side are implemented in kernel space.

Clocking rate limiters The prototype uses software-based token bucket filters to limit the rate of each tunnel. Implementing software rate limiters that work correctly and efficiently at high rates (e.g., 100s of Mbps) requires high precision interrupts, which are not yet available to drivers in all OS/hardware combinations. Instead, we built a high precision clock driven by periodic network events. One core, per rack of servers, stays in a busy loop, and broadcasts a UDP heartbeat packet with the current time to all the servers within that rack once every 0.1ms; the shim layers use these packets to clock their rate limiters. We built a roughly equivalent window-based version of the Seawall shim as proof-of-concept. Windowing is easier to engineer, since it is self-clocking and does not require high precision timers, but incurs the expense of more frequent feedback packets (e.g., once every 10 packets).

Bit-stealing and stateless offload compatibility A practical concern is the need to be compatible with NIC offloads. In particular, adding an extra packet header to support Seawall prevents the use of widely-used NIC offloads, such as large send offload (LSO) and receive side coalescing (RSC) which only work for known packet formats such as UDP or TCP. This leads to increased CPU overhead and decreased throughput. On a quad core 2.66 Intel Core2 Duo with an Intel 82567LM NIC, sending at the line rate of 1Gbps requires 20% more CPU without LSO (net: 30% without vs 10% with LSO) [136].

Though NIC vendors have plans to improve offload support for generic headers, our prototype seeks to maintain performance without depending on such functionality. Seawall maintains the same TCP/IP header format that NIC offload engines expect, without adding any encapsulation headers to store its information. Instead, Seawall *steals* bits from existing packet headers, that is, it encodes information in parts of the packet that are unused or predictable and hence can be restored by the shim at the receiver. For both UDP and TCP, Seawall uses up to 16 bits from the IP ID field, reserving the lower order bits for the segmentation hardware if needed. For TCP packets, Seawall repurposes the timestamp option: it compresses the option Kind and Length fields from 16 bits down to 1 bit, leaving the rest for Seawall data. In virtualized environments, guest OSes are para-virtualized to always include timestamp options. Feedback packets are sent out-of-band in separate packets to reduce encoding demands on bit-stealing. We found bit-stealing easier to engineer than adding extra headers, which could easily lead to performance degradation unless buffers were managed carefully.

Offloading rate limiters and direct I/O Emerging standards to improve network I/O performance, such as direct I/O and SR-IOV, permit guest VMs to bypass the virtual switch and exchange packets directly with the NIC. But, this also bypasses the Seawall shim. Below, we propose a few ways to restore compatibility. However, we note that the loss of the security and manageability features provided by the software virtual switch has limited the deployment of direct I/O NICs in public clouds. To encourage deployment, vendors of such NICs plan to support new features specific to datacenters.

By offloading token bucket- and window-based limiters from the virtual switch to NIC or switch hardware, tenant traffic can be controlled even if guest VMs directly send packets to the hardware. To support Seawall, such offloaded rate limiters need to provide the same granularity of flow classification (entity to entity tunnels) as the shim and report usage and congestion statistics. High end NICs that support stateful TCP, iSCSI, and RDMA offloads already support tens of thousands to millions of window-control engines in hardware. Since most such NICs are programmable, they can likely support the changes needed to return statistics to Seawall. Switch *policers* have similar scale and expressiveness properties. In addition, sidecars can be used to monitor the network for violations (Section 5.1.3). Given the diversity of implementation options, we believe that the design point occupied by Seawall, i.e., using rate- or window-controllers at the network edge, is feasible now and as data rates scale up.

6.4 Evaluation

We ran a series of experiments using our prototype to show that Seawall achieves line rate with minimal CPU overhead, scales to typical data centers, converges to network allocations that are agnostic to communications pattern (i.e., number of flows and destinations) and protocol mix (i.e., UDP and TCP), and provides performance isolation. Through experiments with web workloads, we also demonstrate how Seawall can protect cloud-hosted services against DoS attacks, even those using UDP floods.

All experiments used the token bucket filter-based shim (i.e., rate limiter), which is our best-performing prototype and matches commonly-available hardware rate limiters. The following hold unless otherwise stated: (1) Seawall was configured with the default parameters specified in Section 6.2.6, (2) all results were aggregated from 10 two minute runs, with each datapoint a 15 second average and error bars indicating the 95% confidence interval.

	Throughput (Mb/s)	**CPU @ Sender** (%)	**CPU @ Receiver** (%)
Seawall	947 ± 9	20.7 ± 0.6	14.2 ± 0.4
NDIS	977 ± 4	18.7 ± 0.4	13.5 ± 1.1
Baseline	979 ± 6	16.9 ± 1.9	10.8 ± 0.8

Table 6.1: CPU overhead comparison of Seawall, a null NDIS driver, and an unmodified network stack. Seawall achieved line rate with low overhead.

Testbed For our experiments, we used a 60 server cluster spread over three racks with 20 servers per rack. The physical machines were equipped with Xeon L5520 2.27 GHz CPUs (quad core, two hyperthreads per core), Intel 82576 NICs, and 4GB of RAM. The NIC access links were 1Gb/s and the links from the ToR switches up to the aggregation switch were 10Gb/s. There was no over-subscription within each rack. The ToR uplinks were 1:4 over-subscribed. We chose this topology because it is representative of typical data centers.

For virtualization, we use Windows Server 2008R2 Hyper-V with Server 2008R2 VMs. This version of Hyper-V exploits the Nehalem virtualization optimizations, but does not use the direct I/O functionality on the NICs. Each guest VM was provisioned with 1.5 GB of RAM and 4 virtual CPUs.

6.4.1 Microbenchmarks

Throughput and overhead

To evaluate the performance and overhead of Seawall, we measured the throughput and CPU overhead of tunneling a TCP connection between two machines through the shim. To minimize extraneous sources of noise, no other traffic was present in the testbed during each experiment and the sender and receiver transferred data from and to memory.

Seawall achieved nearly line rate at steady state, with negligible increase in CPU utilization, adding 3.8% at the sender and 3.4% at the receiver (Table 6.1). Much of this overhead was due to the overhead from installing a NDIS filter driver: the null NDIS filter by itself added 1.8% and 2.7% overhead, respectively. The NDIS framework is fairly light weight since it runs in the kernel and requires no protection domain transfers.

Subtracting out the contributions from the NDIS filter driver reveals the overhead due to Seawall, which incurred slightly more overhead on the sender than the receiver. This is expected since the sender does more work: on receiving packets, a Seawall receiver need only buffer congestion information and bounce it back to the sender, while the sender incurs the overhead of rate limiting and may have to merge congestion information from many destinations.

Seawall readily scales to today's data centers. The shim at each node maintains a rate limiter, with a few KBs of state each, for every pair of communicating entities terminating at that node. The per-packet cost on the data path is fixed regardless of data center size. A naive implementation of the rate controller incurs O(DL) complexity per sending entity (VM or task) where D is the number of destinations the VM communicates with and L is the number of links on paths to those destinations. In typical data center topologies, the diameter is small, and serves as an upper bound for L. All network stacks on a given node have collective state and processing overheads that grow at least linearly with D; these dominate the corresponding contributions from the rate controller and shim.

Traffic-agnostic network allocation

Seawall seeks to control the network share obtained by a sender, regardless of traffic. In particular, a sender should not be able to attain bandwidth beyond that allowed by the configured weight, no matter how it varies protocol type, number of flows, and number of destinations.

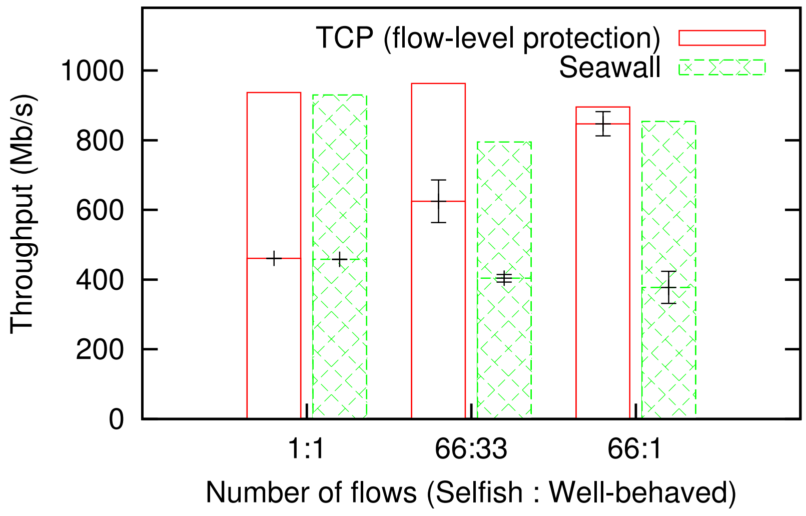
To evaluate the effectiveness of Seawall in achieving this goal, we set up the following experiment. Two physical nodes, hosting one VM each, served as the sources, with one VM dedicated to selfish traffic and the other to well-behaved traffic. One physical node served as the sink for all traffic; it was configured with two VMs, with one VM serving as the sink for well-behaved traffic and the other serving as the sink for selfish traffic.

Both well-behaved and selfish traffic used the same number of source VMs, with all Seawall senders assigned the same network weight. The well-behaved traffic consisted of a single long-lived TCP flow from each source, while the selfish traffic used one of three strategies to achieve a higher bandwidth share: (1) using a full-rate UDP flow, (2) using large numbers of TCP flows, and (3) using many destinations.

Selfish traffic = Full-burst UDP Figure 6.11(a) shows the aggregate bandwidth achieved by the well-behaved traffic (long-lived TCP) when the selfish traffic consisted of full-rate UDP flows. The sinks for well-behaved and selfish traffic were colocated on a node with a single 1Gbps NIC. Because each sender had equal weight, Seawall assigned half of this capacity to each sender. Without Seawall, selfish traffic overwhelms well-behaved traffic, leading to negligible throughput

	TCP victim throughput (Mb/s)
Seawall	429.76
No protection	1.49

(a) Full-rate UDP



(b) Many TCP Flows

Figure 6.11: Seawall ensures that despite using full-rate UDP flows or many TCP flows, the share of a selfish user is held proportional to its weight. (In (b), the bars show total throughput, with the fraction below the divider corresponding to selfish traffic and the fraction above corresponding to well-behaved traffic.)

for well-behaved traffic. By bundling the UDP traffic inside a tunnel that imposed congestion control, Seawall ensured that well-behaved traffic retained reasonable performance.

Selfish traffic = Many TCP flows Figure 6.11(b) shows the bandwidth shares achieved by selfish and well-behaved traffic when selfish senders used many TCP flows. As before, well-behaved traffic ideally should have achieved $\frac{1}{2}$ of the bandwidth. When selfish senders used the same number of flows as well-behaved traffic, bandwidth was divided evenly (left pair of bars). In runs without Seawall, selfish senders that used twice as many flows obtained $\frac{2}{3}$'rds of

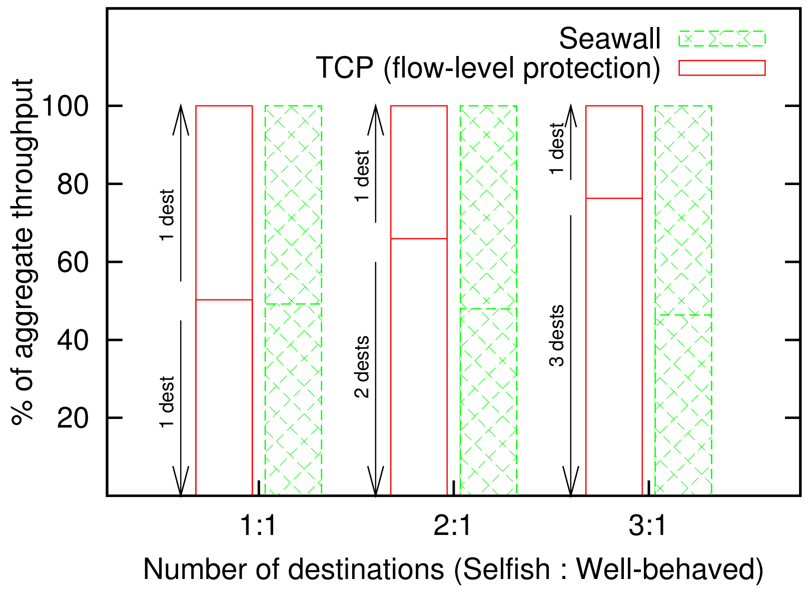


Figure 6.12: By combining feedback from multiple destinations, Seawall ensures that the share of a sender remains independent of the number of destinations it communicates with. (The fraction of the bar below the divider corresponds to the fraction of bottleneck throughput achieved by selfish traffic.)

the bandwidth because TCP congestion control divided bandwidth evenly across flows (middle pair of bars). Runs with Seawall resulted in approximately even bandwidth allocation. Note that Seawall achieved slightly lower throughput in aggregate. This was due to slower recovery after loss– the normal traffic had one sawtooth per TCP flow whereas Seawall had one per source VM; we believe this can be improved using techniques from Section 6.2. When the selfish traffic used 66 times more flows, it achieved a dominant share of bandwidth; the well-behaved traffic was allocated almost no bandwidth (rightmost pair of bars). We see that despite the wide disparity in number of flows, Seawall divided bandwidth approximately evenly. Again, Seawall improved the throughput of well-behaved traffic (the portion above the divider) by several orders of magnitude.

	Throughput (Mb/s)	Latency (s)
Seawall	181	0.61
No protection	157	0.91

Figure 6.13: Despite bandwidth pressure, Seawall ensures that the average HTTP request latency remains small without losing throughput.

Selfish traffic = Arbitrarily many destinations This experiment evaluated Seawall's effectiveness against selfish tenants that opened connections to many destinations. The experiment used a topology similar to that in Figure 6.7. A well-behaved sender VM and a selfish sender VM were located on the same server. Each sink was a VM and ran on a separate, dedicated machine. The well-behaved traffic was assigned one sink machine and the selfish traffic was assigned a variable number of sink machines. Both well-behaved and selfish traffic consisted of one TCP flow per sink. As before, the sending VMs were configured with the same weight, so that well-behaved traffic would achieve an even share of the bottleneck.

Figure 6.12 plots the fraction of bottleneck bandwidth achieved by well-behaved traffic with and without Seawall. We see that without Seawall, the share of the selfish traffic was proportional to the number of destinations. With Seawall, the share of the well-behaved traffic remained constant at approximately half, independent of the number of destinations.

6.4.2 Performance isolation for web servers

To show that Seawall protects against performance interference, we evaluated the achieved level of protection against a DoS attack on a web server. Since cloud datacenters are often used to host web-accessible services, this is a common use case.

In this experiment, an attacker targeted the HTTP responses sent from the web server to its clients. To launch such attacks, an adversary places a source VM and a sink VM such that traffic between these VMs crosses the same bottleneck links as the web server. The source VM is close to the server, say on the same rack or machine, while the sink VM is typically on another rack. Depending on where the sink is placed, the attack can target the ToR uplink or another link several hops away.

All machines were colocated on the same rack. The web server VM, running Microsoft IIS 7, and attacker source VM, generating UDP floods, resided in separate, dedicated physical machines. A single web client VM requested data from the server and shared a physical machine with an attacker sink VM. The web clients used WcAsync to generate well-formed web sessions. Session arrivals followed a Poisson process and were exponentially sized with a mean of 10 requests. Requests followed a WebStone distribution, varying in size from 500B responses to 5MB responses with smaller files being much more popular.

As expected, a full-rate UDP attack flood caused congestion on the access link of the web client, reducing throughput to close to zero and substantially increasing latency. With Seawall, the web server behaved as if there were no attack. To explore data points where the access link was not overwhelmed, we dialed down the UDP attack rate to 700Mbps, enough to congest the link but not to stomp out the web server's traffic. While achieving roughly the same throughput as in the case of no protection, Seawall improved the latency observed by web traffic by almost 50% (Figure 6.13). This is because sending the attack traffic through a congestion controlled tunnel ensured that the average queue size at the bottleneck stays small, thereby reducing queuing delays.

6.5 Discussion

Here, we discuss how Seawall can be used to implement rich cloud service models that provide bandwidth guarantees to tenants, the implications of our architectural decisions given trends in data centers and hardware, and the benefits of jointly modifying senders and receivers to achieve new functionality in data center networks.

6.5.1 Sharing policies

Virtual Data Centers (VDCs) have been proposed [89, 78, 141] as a way to specify tenant networking requirements in cloud data centers. VDCs seek to approximate, in terms of security isolation and performance, a dedicated data center for each tenant and allows tenants to specify SLA constraints on network bandwidth at per-port and per-source/dest-pair granularities. When allocating tenant VMs to physical hardware, the data center fabric simultaneously satisfies the specified constraints while optimizing node and network utilization.

Though Seawall policies could be seen as a simpler-to-specify alternative to VDCs that closely matches the provisioning knobs (e.g., disk, CPU, and memory size) of current infrastructure clouds, Seawall's weight-based policies can enhance VDCs in several ways. Some customers, through analysis or operational experience, understand the traffic requirements of their VMs; VDCs are attractive since they can exploit such detailed knowledge to achieve predictable performance. To improve VDCs with Seawall, the fabric uses weights to implement the hard bandwidth guarantees specified in the SLA: with appropriate weights, statically chosen during node- and path-placement, Seawall will converge to the desired allocation. Unlike implementations based on static reservations [78], the Seawall implementation is work-conserving, max-min fair, and achieves higher utilization through statistical multiplexing.

Seawall also improves a tenant's control of its own VDC. Since Seawall readily accepts dynamic weight changes, each tenant can adjust its allocation policy at a fine granularity in response to changing application needs. The fabric permits tenants to reallocate weights between different tunnels so long as the resulting weight does not exceed the SLA; this prevents tenants from stealing service and avoids having to rerun the VM placement optimizer.

6.5.2 System architecture

Topology assumptions The type of topology and available bandwidth affects the complexity requirements of network sharing systems. In full bisection bandwidth topologies, congestion can only occur at the core. System design is simplified [155, 141, 123], since fair shares can be computed solely from information about edge congestion, without any topology information or congestion feedback from the core.

Seawall supports general topologies, allowing it to provide benefits even in legacy or cost-constrained data centers networks. Such topologies are typically bandwidth-constrained in the core; all nodes using a given core link need to be accounted for to achieve fair sharing, bandwidth reservations, and congestion control. Seawall explicitly uses topology information in its control layer to prevent link over-utilization.

Rate limiters and control loops Using more rate limiters enables a network allocation system to support richer, more granular policies. Not having enough rate limiters can result in aliasing. For instance, VM misbehavior can cause Gatekeeper [141] to penalize unrelated VMs sending to the same destination. Using more complex rate limiters can improve system performance. Rate limiters based on multi-queue schedulers such as DWRR or Linux's hierarchical queuing classes can utilize the network more efficiently when rate limiter parameters and demand do not match, and the self-clocking nature of window-based limiters can reduce switch buffering requirements as compared to rate-based limiters. However, having a large number of complex limiters can constrain how a network sharing architecture can be realized, since NICs and switches do not currently support such rate limiters at scale.

To maximize performance and policy expressiveness, a network allocation system should support a large number of limiters of varying capability. The current Seawall architecture can support rate- and window-based limiters based in hardware and software. As future work, we are investigating ways to map topology information onto hierarchical limiters; to compile policies given a limited number of available hardware limiters; and to tradeoff rate limiter complexity with controller complexity, using longer adaptation intervals when more capable rate limiters are available.

6.5.3 Partitioning sender/receiver functionality

Control loops can benefit from receiver-side information and coordination, since the receiver is aware of the current traffic demand from all sources and can send feedback to each with lower overhead. Seawall currently uses a receiver-driven approach customized for map-reduce to achieve better network scheduling; as future work we are building a general solution at the shim layer.

In principal, a purely receiver-directed approach to implementing a new network allocation policy, such as that used in [155, 141], might reduce system complexity since the sender TCP stack does not need to be modified. However, virtualization stack complexity does not decrease substantially, since the rate controller simply moves from the sender to the receiver. Moreover, limiting changes to one endpoint in data centers provides little of the adoption cost advantages found in the heterogeneous Internet environment. Modifying the VMs to defer congestion control to other layers can help researchers and practitioners to identify and deploy new network sharing policies and transport protocols for the data center.

A receiver-only approach can also *add* complexity. While some allocation policies are easy to attain by treating the sender as a black box, others are not. For instance, eliminating fatesharing from Gatekeeper and adding weighted, fair work-conserving scheduling appears non-trivial. Moreover, protecting a receiver-only approach from attack requires adding a detector for non-conformant senders. While such detectors have been studied for WAN traffic [63], it is unclear whether they are feasible in the data center. Such detectors might also permit harmful traffic that running new, trusted sender-side code can trivially exclude.

CHAPTER 7

RELATED WORK

This chapter discusses the contributions of this dissertation within the context of prior work.

7.1 Network architecture

Recent work has re-examined the traditional architecture of the network layer. Where the basic functionality of switches and routers was traditionally divided into the data- and control-planes, recent work has proposed adding a knowledge plane [46] to facilitate coordination between network participants. To our knowledge, NetQuery is the first realization of a knowledge plane for federated networks comprising mutually-untrusting administrative domains.

Other recent work has proposed new programming platforms for building the control plane. Traditionally, new control plane functionality has been built by extending or defining new network control protocols to coordinate control processes running independently on every switch and router. That approach requires invasive upgrades to each device to add new functionality; design is also challenging since each new protocol solves the distributed control problem anew. Recent work on software-defined networks (SDN) addresses the design and deployment of new functionality by (1) exposing a programming model based on a centralized view of the network state and (2) providing reusable abstractions [32, 72, 74, 97]. The centralized model is conceptually simpler and requires only changing a few controllers, as opposed to every element in the network.

Although some of the network guarantees provided by NetQuery applications had been proposed in prior work, much of that prior work has been implemented with one-off extensions to the control and data plane. In contrast, NetQuery's knowledge plane is a common abstraction from which one can build such guarantees; this abstraction could be integrated into SDN platforms. The knowledge plane is a representation of global system state, which some SDN platforms already maintain and expose to programmers. On such platforms, the incremental cost of adding a knowledge plane is reduced since the costs of maintaining this global system state and training systems builders to use it are amortized between the control plane and knowledge plane.

NetQuery complements the data plane work on establishing network guarantees by providing standard abstractions for disseminating and reasoning about such guarantees. The remainder of this section describes in more detail these categories of prior work.

7.1.1 Control platforms with logically-centralized system state

The global view of network state that is maintained by a knowledge plane closely resembles the global view maintained by logically-centralized control platforms for networks. These network control platforms include network exception handlers (NEH) [92], ident++ [113], Maestro [33], NOX [74], Onix [97], and DECOR [39]. All are logically centralized systems for single-operator networks that disseminate network information to administrative applications. Similar techniques are also found in control platforms for cloud data centers [114, 98].

NEH collects dynamic topology, load statistics, and link costs from the network infrastructure, and exposes these properties so that end hosts can detect and react to exceptional network conditions. ident++ collects metadata about flows from end hosts to enable richer access control policies. Maestro, NOX, and DECOR are control platforms for running network control plane or management applications on a central controller. Onix leverages distributed systems techniques to build a central controller in a scalable, reliable fashion.

NetQuery collects similar information (e.g., adjacency and forwarding tables) about network devices and makes this information available in a similar fashion; indeed, its tuple-based representation and schemas, while independently developed, is similar to that of the Onix NIB [97] and CMIS objects [86]. NetQuery supports such network management applications for enterprise networks, while also supporting applications that issue queries spanning multiple ASes. The NetQuery knowledge plane provides a richer data representation that can support heterogeneous information sources by tracking the source of every statement and by leveraging trusted hardware.

Declarative programming [159] has been proposed as an alternative for building applications and for managing networks. In such systems, application logic is written as high-level rules that manipulate a database representation of the network. DECOR [39] uses declarative programming to autoconfigure network devices to meet high level operational goals. DECOR uses logical specifications for device semantics and state that can be helpful in writing NetQuery analyses and sanitizers. Both NetQuery and DECOR provide frameworks for extending existing devices to support policy analysis and management applications. NetQuery applications can span mutually distrusting administrative domains, for which we provide trust establishment and sanitization techniques not needed in the problem domain of DECOR.

7.1.2 Establishing network guarantees

Leveraging the TPM

Trusted Network Connect [145] and attestation-based policy enforcement [131] are network access control systems with similar client authorization to NetQuery. Before an end host is allowed to join the network, these systems use attestation to verify that the end host's software and hardware configuration satisfies the access policy. Unlike NetQuery, these systems do not provide a channel by which end hosts can discover the properties of a network before deciding to connect.

The ubiquity of TPMs has inspired many systems that rely on trusted end host functionality to improve network security and performance. Such work uses trusted end host functionality to provide local guarantees. For instance, ETTM moves middlebox and filtering to end hosts [52], while other work relies on end hosts to perform packet classification [25, 125, 76]. NetQuery provides a standard interface for describing these local guarantees, enabling other applications to use them. Several of these systems rely on network-wide reasoning to provide guarantees to a single administrative domain; NetQuery can be used to perform such reasoning and generalizes it to provide guarantees to multiple domains.

Data-plane and monitoring techniques

Several extensions to IP to provide guarantees about the sender of each packet have been proposed. Accountable IP (AIP) [17], [29], and packet passports [103] provide accountability for network packets. These systems use optimizations whose safety rely on global network configuration invariants spanning multiple ASes. They can use NetQuery to verify these trust assumptions. Assayer [119] attaches trustworthy sender information to packets as unforgeable annotations, obviating the need to reconstruct this information at middleboxes [120]. Network witness [62] uses attestation to enable TPM-equipped end hosts to serve as trusted network monitoring points. NetQuery, Assayer, and Network witness make similar use of the TPM. Network Confessional [20] provides verifiable performance measurements at the granularity of network paths and peering points.

Overall, such data plane approaches are complementary to a knowledge plane; factoids extracted through these techniques can be extracted and disseminated through NetQuery.

Control plane techniques

sBGP [95], BIND [135], and Whisper [142] extend BGP, a control plane protocol, with additional cryptographic protections. BIND and Whisper use cryptography to chain messages together to allow participants to detect route manipulation. BIND leverages the TPM to inductively attest to the validity of route advertisements; the inter-message chaining is bootstrapped from attestation certificates. A route advertisement is valid if it comes from the owner of a prefix or if it is created by a TPM-attested transformation on route advertisements. Attestation relies on a PKI; Whisper avoids PKI dependencies by relaxing the provided guarantees and exploiting network structure. Whisper enables a recipient to check whether the set of routes to a destination, as received from different next-hop routers, is consistent; an attacker can only subvert this check if it compromises a cut of the network.

By comparison, NetQuery analysis and information dissemination is separated from the control protocol; the few control plane changes to support NetQuery abstractions can be exposed as a knowledge plane and amortized across many applications. Several systems for establishing new guarantees share this separation from control plane protocols.

NetReview [79] enforces fault detection for BGP behavior using a tamper-evident log. Since NetQuery supports analysis over router RIBs, it can check similar BGP policies as NetReview. NetReview relies on an AS's neighbors to achieve tamper-evidence and trustworthy detection on publicly-disclosable information. In contrast, NetQuery can bootstrap trust from TPMs where available, and it can also use sanitizers to perform trustworthy analysis on confidential network information. Like Whisper, NetReview avoids PKI by relying on network structure assumptions about how logs are collected and disseminated within the network.

The verification primitives that a knowledge plane provides can impact its compatibility with applications and its deployment assumptions. If NetReview's primitives were used to build a knowledge plane, that knowledge plane will not require a PKI or TPMs, but can only support applications that rely on locally-checkable guarantees. NetQuery's attribution and attestation-based dissemination structure can impose fewer restrictions on application structure, since, confidentiality constraints aside, it can propagate information anywhere in the network. Likewise, one could construct knowledge planes based on the dissemination primitive of protocols like Whisper or Byzantine-robust routing [121]; these would give rise to yet another class of supported applications and underlying networks.

Keller et al [94] applies trustworthy computing techniques to an ISP operating model where service providers build wide area services using virtual router slices leased from infrastructure providers. This system uses trusted hardware, keys hidden from infrastructure providers, and sample-based tracing to verify that the infrastructure provider has properly installed the slices. While NetQuery applies similar techniques, NetQuery targets the existing ISP operating model and uses a logical framework for analysis.

Comparing providers

Since many applications are sensitive to network performance, verifying claims about the network is a key concern. Many data-plane techniques for monitoring network performance in the Internet have been proposed [158, 20]; such techniques could be applied to cloud data center networks

Several industry efforts have focused on promoting a transparent, competitive market for cloud computing. These include work on improving standardization, portability, and schemas for specifying computation [98]; creating markets for spare cloud capacity [57]; defining schemas for describing and assisting in audit [84]; and comparing the performance characteristics of cloud services to aid in provider selection [102]. Our knowledge plane work for cloud data centers was motivated by and complements such efforts.

In general, such measurements could be collected by the knowledge plane in a remotely verifiable fashion and disseminated as a public service, with the control and decision functionality implemented as NetQuery applications.

7.2 Establishing resource allocation and performance isolation

Like Seawall, Congestion Manager aggregates congestion feedback from multiple flows to drive shared control loops and provides similar application interfaces for querying congestion control state [26]. But because its feedback aggregation operates on a per-destination basis, it has similar shortcomings to the rejected approach from Section 6.2.2. The deployment and trust model also differ; while Congestion Manager enables Internet end hosts and applications to voluntarily improve performance while preserving TCP-friendly behavior, Seawall leverages the homogeneity and code isolation properties of data center networking stacks to transparently enforce global allocation policies on network capacity.

Proportional allocation of shared resources has been a recurring research topic in operating systems and virtualization [150, 75]. To the best of our knowledge, Seawall is the first to extend this to the data center network and support generic sending entities (VMs, applications, tasks, processes, etc.). Multicast congestion control [69], while similar at first blush, targets a very different problem since they have to allow for any participant to send traffic to the group while ensuring TCP-friendliness. It is unclear how to adapt these schemes to proportionally divide the network.

Recent work in hypervisor, network stack, and software routers have shown that software-based network processing, like that used in Seawall for monitoring and rate limiting, can be more flexible than hardware-based approaches yet achieve high performance. [132] presents an optimized virtualization stack that achieves comparable performance to direct I/O. The Sun Crossbow network stack provides an arbitrary number of bandwidth-limited virtual NICs [144]. Crossbow provides identical semantics regardless of underlying physical NIC and transparently leverages offloads to improve performance. Seawall's usage of rate limiters can benefit from these ideas.

QCN is an emerging Ethernet standard for congestion control in datacenter networks [117]. In QCN, upon detecting a congested link, the switch sends feedback to the heavy senders. The feedback packet uniquely identifies the flow and congestion location, enabling senders that receive feedback to rate limit specific flows. QCN uses explicit feedback to drive a more aggressive control loop than TCP. While QCN can throttle the heavy senders, it is not designed to provide fairness guarantees, tunable or otherwise. Further, QCN requires changes to switch hardware and can only cover purely Layer 2 topologies.

Fair queuing mechanisms in switches has long been studied [51]. Link local sharing mechanisms, such as Weighted Fair Queuing and Deficit Round Robin, separate traffic into multiple queues at each switch port and arbitrate service between the queues in some priority or proportion. NetShare [100] builds on top of WFQ support in switches. This approach is useful to share the network between a small number of large sending entities (e.g., a whole service type, such as "Search" or "Distributed storage" in a platform data center). The number of queues available in today's switches, however, is several orders of magnitude smaller than the numbers of VMs and tasks in today's datacenters. More fundamentally, since link local mechanisms lack end-to-end information they can let significant traffic through only to be dropped at some later bottleneck on the path. Seawall can achieve better scalability by mapping many VMs onto a small, fixed number of queues and achieves better efficiency by using end-to-end congestion control.

The suitability of a resource allocation scheme for a given application and system architecture varies depending on the resource allocation scheme's underlying trust assumptions. In multi-tenant settings, neither traffic sources nor destinations can be trusted to implement the allocation scheme; hence other components must be trusted to provide resource allocation guarantees. Seawall, along with the preceding schemes, rely on extending trusted data plane elements, such as the hypervisor, routers, or switches, with new, stateful queues and rate limiters. SideCar provides a less disruptive, lower overhead transition path: while the data plane elements must still be trusted, they only need to export a sampling or redirection primitive to support new resource allocation schemes. Trickles [137] is a stateless bandwidth allocation scheme that, unlike other schemes, can enforce TCP-friendly, congestion controlled bandwidth allocation while requiring substantially less state in the trusted data plane elements. By using parallelizable, lightweight cryptographic and state compression techniques, Trickles can enforce such allocations across many simultaneous flows at line rate.

CHAPTER 8

CONCLUSION

Federation is an important technique for building large scale, geographically-distributed computing infrastructure. In deployed systems, such as the Internet, and in proposal systems, such as new cloud computing infrastructure, federation arises due to competitive pressures between service providers and can improve the economic feasibility of the system. But different providers in a federated infrastructure can vary widely in performance, robustness, and security, potentially affecting the correctness of an application that relies on a disparate set of infrastructure providers.

8.1 Challenges and contributions

This thesis investigated techniques to improve coordination in federated systems and to expand the guarantees that they can provide to applications. Doing so ensures that applications can determine how to best accommodate and exploit the inherent diversity among infrastructure providers. Once applications can make better-informed decisions on how to use the infrastructure, they can benefit from assurances about correctness, improved performance, and reduced deployment costs.

8.1.1 Knowledge planes for federated distributed systems

In general, the infrastructure dependencies of an application will span several administrative domains. This dissertation presented NetQuery, a system that streamlines coordination between applications and administrative domains by providing reasoning abstractions that not only enable applications to infer characteristics of interest about a single administrative domain but also to make inferences that span a federated system. By running NetQuery analysis over a logical representation of a system, applications can verify that the system satisfies requirements, improving the assurance provided by existing applications and enabling new kinds of applications that are difficult to build with only the ad hoc reasoning mechanisms provided by current infrastructure.

NetQuery disseminates information about participating networks through a knowledge plane that stores a logical representation of a federated system. Applications can query the knowledge plane to fetch system information for analysis. All administrative domains cooperate in building the knowledge plane. Each administrative domain deploys infrastructure to describe their own local portion of the federated system; the combined logical representation spans the entire federated system. The logical representation includes information about the current topology, configuration, and state; NetQuery analyses use this knowledge plane information to logically infer characteristics that hold in the network.

The key challenges in building NetQuery are (1) ensuring that applications reason using trustworthy, meaningful information and (2) preserving the confidentiality of proprietary operator information. Addressing the former assures that the characteristics inferred by applications are sound and actually hold in the real world while the latter facilitates deployment by ensuring that the knowledge plane respects the existing policies and trust relationships of network operators.

Contributions

NetQuery directly addresses these challenges as follows.

Establishing the trustworthiness of knowledge plane information To aid in establishing trust, NetQuery tracks attribution information for all information in the knowledge plane. For every piece of information, NetQuery records the principal (e.g., device, process, or network operator) that was responsible for inserting it. Each application can reason using only information from trustworthy sources. NetQuery does not impose a global trust policy; rather, applications are free to independently choose an appropriate trust policy.

Information is trustworthy if there is sufficient reason to expect that the logical view correlates to the real-world system. Attribution information implicitly discourages lying: should an application discover a lie, it can filter subsequent information from that source, and the network is strictly more trustworthy than it would have been without the techniques proposed in this thesis. NetQuery also provides two explicit, interrelated mechanisms for providing this assurance. First, attestation serves as an authentication mechanism to verify that information comes from known-good hardware and software platforms. Second, operators might try to fool attested platforms into exporting information that diverges from the real world; to prevent this, known-good platforms can be equipped with protection and monitoring mechanisms to guard their inputs and outputs.

Preserving confidentiality of operator information To address confidentiality concerns, each operator participating in NetQuery can specify an access policy for the information that it exports to the knowledge plane. Since operators will likely specify knowledge plane access policies that match their existing policies, which tightly restrict what can be revealed to outside parties, NetQuery provides a sanitizer abstraction that enables analyses to execute without disclosing such information. Rather than exporting this information directly to external applications, which increases the TCB size for maintaining confidentiality, information is disclosed only to sanitizer nodes that are trusted by both the operator and external party.

Integration with existing systems The design of NetQuery employs several architectural choices and incremental deployment strategies to facilitate its introduction to existing federated systems.

To support an expressive, widely-applicable reasoning engine that can infer a wide range of system characteristics, NetQuery can incorporate information from a broad population of devices and provides in-depth information about each class of devices.

NetQuery can bootstrap trust in a broad population of devices in a scalable, low cost manner using the attestation primitives of trusted hardware, such as the TPM. This bootstrap process involves identifying a hardware and software platform then using attestation to bind NetQuery attribution information to this platform. The costs of this process are favorable: (1) validating the platform is a fixed engineering cost that is amortized across the many individual units of a particular type of device and (2) embedding TPMs adds minimal cost to each unit. The per-operation overhead from attestations is low since each attestation can be used across many NetQuery operations.

NetQuery's in-depth information comes from leveraging the information that is already available from existing interfaces for system management, such as SNMP. Such interfaces are widely deployed on devices and export considerable information about those devices. NetQuery can make such information suitable for analysis by adding attribution and monitoring mechanisms to verify that the reported information matches with the real network.

8.1.2 Establishing guarantees in data center networks

As the computational demands of Internet services increases, data center networks are increasingly becoming a potential source of bottlenecks and increases to system cost. At the same time, the largely homogeneous environment within each data center presents unique opportunities and challenges not present in heterogeneous networks such as the Internet.

So that management and provisioning can scale efficiently, many large-scale data centers use centralized, unified control over the entire infrastructure, including network devices, compute nodes, and end host software stacks. This close integration between networking and compute nodes provides opportunities for building new networking abstractions.

Adding a general-purpose programming model for the network can facilitate the deployment of such functionality. But many programming models, such as those that support per-packet processing, substantially increase network cost;

packet processing architectures with high costs would be difficult to deploy in cost-sensitive cloud data centers. More programmability could also lead to greater variation in network capabilities across different data center providers; applications would benefit from improved signaling of these differences.

Many data center networks, such as those for large-scale data centers for public cloud computing providers, have heterogeneous, sometimes malicious, workloads. Such workloads are inherent to public cloud data centers due to their use of statistical multiplexing to improve efficiency and high bandwidth links to applications. In addition to these benefits, such mixing of workloads also raises potential DoS attacks and performance interference. Thus, the cloud environment poses challenges to porting legacy applications from dedicated infrastructure with fewer, better controlled concurrent workloads, and to deploying new applications with critical performance and security requirements.

Contributions

This dissertation addressed these challenges with the following contributions:

Trusted packet processors We proposed a programming model based on trusted packet processors. Trusted packet processors achieve generality by allowing applications to write programs that distribute custom processing anywhere in the network. Trusted packet processors are placed ubiquitously throughout the network topology: by using trusted packet processors embedded in end host network stacks and switches, new applications can leverage custom packet processing and monitoring from the network edge to core.

Trusted packet processors leverage trustworthy computing techniques to address the increased complexity of reasoning about the properties of programmable networks. Trusted packet processors execute packet processing and monitoring code in an attested environment that is isolated from user applications. Attestations provide users and developers of new network programs with the assurance that packet processing code implementing a global network program has been properly installed onto network devices. Combined with isolation, which protects the installed programs from malicious applications, trusted packet processors provide the assurance that the programs and their associated guarantees and capabilities are in effect. Data centers can use NetQuery to signal such assurances to other network participants.

To scale to the data rates found in typical data centers at a competitive hardware cost to existing networks, in-network trusted packet processors restrict the volume of traffic that network programs can process. This design decision reduces the processing demands on switches and sends the majority of traffic over hardware-efficient dedicated processing paths. The resulting processing model is well-suited for implementing sampling-based techniques; we proposed several design patterns for building new network guarantees using this approach.

Seawall bandwidth allocator This dissertation presented the design and implementation of the Seawall bandwidth allocator, which provides data center operators with better control over network bandwidth allocation in their networks. Seawall is designed for the link speeds, high churn, and diverse workloads of cloud data centers.

To achieve scalability and efficient handling of churn, Seawall computes allocations using distributed, end-to-end congestion control. Since this algorithm is implemented in compute nodes, it can scale to larger numbers of communicating nodes and larger policies than techniques based on network hardware. Standard congestion control loops, like that of TCP, are highly sensitive to the specific communications patterns; by opening up many flows, a malicious or selfish tenant can consume a large fraction of network capacity. By combining the feedback from multiple flows and charging the indicated losses to links, rather than flows, Seawall achieves bandwidth allocations that are agnostic to variations in the pattern of flows.

8.2 Final remarks

This dissertation proposes new interfaces and mechanisms for satisfying and establishing application guarantees in federated systems. The knowledge plane interface enables participants in federated systems to convince one another that a given system satisfies some guarantee of interest to application, given the configuration and construction of a system. Experiments show that NetQuery performs well on real routers and can support the volume of network events found in large enterprise and ISP networks. Overall, NetQuery's extensible data model and flexible logic supports a diverse range of applications and can help ISPs and cloud operators differentiate their services. We believe that NetQuery's design principles and abstractions address significant obstacles to building a practical knowledge plane for federated systems and enable novel applications based on global system reasoning. In particular, NetQuery addresses long-standing obstacles to inter-domain coordination on the Internet. NetQuery also represents a critical first step to commoditizing digital resources in the cloud beyond a single provider and enabling multiple providers to create dynamic federations of cloud resources.

The trusted packet processor and Seawall mechanisms allow data center network operators to establish new kinds of guarantees about their systems. Through a design and feasibility study, we showed that trusted packet processors can enable network programs to control and observe traffic at all points in a network while maintaining compatibility with existing switches and incurring similar costs. Experiments on a full implementation demonstrated that Seawall protects against DoS attacks and improves control over bandwidth allocation and does so while achieving line rates with minimal overhead. Together, these techniques facilitate efficient resource utilization and enable the construction of new control- and data-plane abstractions.

BIBLIOGRAPHY

[1] Nova Database Schema.
`http://wiki.openstack.org/NovaDatabaseSchema`.

[2] The Quagga routing suite. `http://www.quagga.net/`.

[3] *CIM Concepts White Paper*. Distributed Management Task Force, Inc, June 2003.

[4] *IEEE Standard for Information technology– Telecommunications and information exchange between systems– Local and metropolitan area networks– Specific requirements. Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications– Amendment 6: Medium Access Control (MAC) Security Enhancements*. IEEE Computer Society, July 2004. IEEE Std 802.11i-2004.

[5] *IEEE Standard for Local and metropolitan area network– Media Access Control (MAC) Security*. IEEE Computer Society, August 2006. IEEE Std 802.1AE-2006.

[6] *Understanding the Available Bit Rate (ABR) Service Category for ATM VCs*. Cisco Systems, June 2006.

[7] *Fibre Channel: Backbone - 5*. American National Standard for Information Technology, June 2009.

[8] *Open Virtualization Format Specification*. Distributed Management Task Force, Inc, January 2010.

[9] *Payment Card Industry (PCI) Data Security Standard v.2.0*. Payment Card Industry: Security Standards Council, October 2010.

[10] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A Scalable, Commodity Data Center Network Architecture. In *Proceedings of ACM SIGCOMM*, Seattle, Washington, August 2008.

[11] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, and Jitendra Padhye. Data Center TCP (DCTCP). In *Proceedings of ACM SIGCOMM*, New Delhi, India, August 2010.

[12] Amazon.com. Amazon EC2 Dedicated Instances.
`http://aws.amazon.com/ec2/dedicated-instances/`.

[13] Amazon.com. Amazon EC2 Pricing.
`http://aws.amazon.com/ec2/pricing/`.

[14] Amazon.com. Amazon EC2 Reserved Instances.
`http://aws.amazon.com/ec2/reserved-instances/`.

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

[16] Ganesh Ananthanarayanan, Srikanth Kandula, Albert Greenberg, Ion Stoica, Yi Lu, Bikas Saha, and Ed Harris. Reining in the Outliers in MapReduce Clusters Using Mantri. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, British Columbia, October 2010.

[17] David G. Andersen, Hari Balakrishnan, Nick Feamster, Teemu Koponen, Daekyeong Moon, and Scott Shenker. Accountable Internet Protocol. In *Proceedings of ACM SIGCOMM*, Seattle, Washington, August 2008.

[18] AOL. AOL Transit Data Network: Settlement-Free Interconnection Policy, 2006.
`http://www.atdn.net/settlement\%5Ffree\%5Fint.shtml`.

[19] David Applegate, Aaron Archer, Vijay Gopalakrishnan, Seungjoon Lee, and K. K. Ramakrishnan. Optimal Content Placement for a Large-scale VoD System. In *Proceedings of ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Philadelphia, Pennsylvania, December 2010.

[20] Katerina Argyraki, Petros Maniatis, and Ankit Singla. Verifiable Network-Performance Measurements. In *Proceedings of ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Philadelphia, Pennsylvania, December 2010.

[21] Michael Armbrust, Amando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, Ion Stoica, and Others. Above the clouds: A Berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS, University of California, Berkeley, February 2009.

[22] Todd W. Arnold and Leendert Van Doorn. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eServer. *IBM Journal of Research and Development*, 48(3/4):491–503, May 2004.

[23] AT&T. AT&T Internet Protection Service, August 2009. `http://www.corp.att.com/abs/serviceguide/docs/ip_sg.doc`.

[24] Ioannis Avramopoulos and Jennifer Rexford. Stealth Probing: Efficient Data-Plane Security for IP Routing. In *Proceedings of USENIX Annual Technical Conference*, Boston, Massachusetts, May 2006.

[25] Kwang-Hyun Baek and Sean W. Smith. Preventing Theft of Quality of Service on Open Platforms. In *Proceedings of IEEE/CREATE-NET Workshop on Security and QoS in Communication Networks*, Athens, Greece, September 2005.

[26] Hari Balakrishnan, Hariharan Rahul, and Srinivasan Seshan. An Integrated Congestion Management Architecture for Internet Hosts. In *Proceedings of ACM SIGCOMM*, Cambridge, Massachusetts, August 1999.

[27] Andrew Begel, Steven McCanne, and Susan L. Graham. Bpf+: Exploiting Global Data-flow Optimization in a Generalized Packet Filter Architecture. *ACM SIGCOMM Computer Communication Review (CCR)*, 29(4):123–134, October 1999.

[28] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, British Columbia, October 2010.

[29] Adam Bender, Neil Spring, Dave Levin, and Bobby Bhattacharjee. Accountability as a Service. In *Proceedings of Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, Santa Clara, California, June 2007.

[30] Karl Bode. Why Are ISPs Still Advertising Limited Services As Unlimited?, December 2008. `http://www.dslreports.com/shownews/Why-Are-ISPs-Still-Advertising-Limited-Services-As-Unlimited-99769`.

[31] Randy Bush. An Operational ISP and RIR PKI, April 2006.

https://www.arin.net/participate/meetings/reports/ARIN_XVII/PDF/sunday/pki-bush.pdf.

[32] Matthew Caesar, Donald Caldwell, Nick Feamster, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. Design and Implementation of a Routing Control Platform. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, May 2005.

[33] Zheng Cai, Florin Dinu, Jie Zheng, Alan L. Cox, and T.S. Eugene Ng. The Preliminary Design and Implementation of the Maestro Network Control Platform. Technical Report TR08-13, Rice University, October 2008.

[34] Martin Casado, Pei Cao, Aditya Akella, and Niels Provos. Flow-Cookies: Using Bandwidth Amplification to Defend Against DDoS Flooding Attacks. In *Proceedings of IEEE International Workshop on Quality of Service (IWQoS)*, New Haven, Connecticut, June 2006.

[35] J. Case, M. Fedor, M. Schoffestall, and J. Davin. Simple Network Management Protocol, May 1990. IETF RFC 1157.

[36] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, November 1998.

[37] Ronnie Chaiken, Bob Jenkins, Perke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and Efficient Parallel Processing of Massive Data Sets. In *Proceedings of International Conference on Very Large Data Bases (VLDB)*, Auckland, New Zealand, August 2008.

[38] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, and Wilson C. Hsieh. BigTable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2), June 2008.

[39] Xu Chen, Yun Mao, Z. Morley Mao, and Jacobus Van der Merwe. DECOR: DEClarative network management and OpeRation. *ACM SIGCOMM Computer Communication Review (CCR)*, 40(1):61–66, January 2010.

[40] Pau Cheng, Rohatgi Pankaj, Claudia Keser, Paul A. Karger, Grant M. Wagner, and Angela Reninger. Fuzzy Multi-Level Security: An Experiment on Quantified Risk-Adaptive Access Control. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, May 2007.

[41] Ludmila Cherkasova, Diwaker Gupta, and Amin Vahdat. Comparison of the three CPU schedulers in Xen. In *Proceedings of ACM SIGMETRICS*, San Diego, California, June 2007.

[42] Yang-hua Chu, Sanjay G. Rao, and Hui Zhang. A Case for End System Multicast. In *Proceedings of ACM SIGMETRICS*, Santa Clara, California, June 2000.

[43] Cisco Systems. Cisco TrustSec. `http://www.cisco.com/en/US/netsol/ns1051/index.html`.

[44] Cisco Systems. IP SLAs–LSP Health Monitor, June 2006. `http://www.cisco.com/en/US/docs/ios/12_4t/12_4t11/ht_hmon.html`.

[45] Benoit Claise. Cisco Systems NetFlow Services Export Version 9, October 2004. IETF RFC 3954.

[46] David D. Clark, Craig Partridge, J. Christopher Ramming, and John T. Wroclawski. A Knowledge Plane for the Internet. In *Proceedings of ACM SIGCOMM*, Karlsruhe, Germany, August 2003.

[47] Bill Claybrook. Comparing cloud risks and virtualization risks for data center apps. `http://searchdatacenter.techtarget.com/tip/0,289483,sid80\_gci1380652,00.html`.

[48] Danny Collins. Is Your Provider Truly Multi-Homed? `http://www.webmasterjoint.com/webmaster-articles/web-hosting/10-multi-homed-is-your-provider-truly-multi-homed.html`.

[49] Jon Crowcroft and Philippe Oechslin. Differentiated end-to-end Internet services using a weighted proportional fair sharing TCP. *ACM SIGCOMM Computer Communication Review (CCR)*, 28(3), August 1998.

[50] Steve E. Deering and David R. Cheriton. Multicast routing in datagram internetworks and extended LANs. *ACM Transactions on Computer Systems (TOCS)*, 8(2):85–110, May 1990.

[51] Alan Demers, S. Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review (CCR)*, 19(4), 1989.

[52] Colin Dixon, Arvind Krishnamurthy, and Tom Anderson. An End to the Middle. In *Proceedings of USENIX HotOS*, Monte Verite, Switzerland, May 2009.

[53] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, and Byung-Gon Chun. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, Montana, October 2009.

[54] Yaozu Dong, Zhao Yu, and Greg Rose. SR-IOV Networking in Xen: Architecture, Design and Implementation. In *Proceedings of USENIX Workshop on I/O Virtualization (WIOV)*, San Diego, California, December 2008.

[55] Nick Duffield and Matthias Grossglauser. Trajectory sampling for direct traffic observation. *IEEE/ACM Transactions on Networking (TON)*, 9(3):280–292, June 2001.

[56] Paul England, Butler Lampson, John Manferdelli, Marcus Peinado, and Bryan Willman. A Trusted Open Platform. *Computer*, 36(7):55–62, July 2003.

[57] enomaly. SpotCloud - Cloud Capacity Clearing House. `http://www.spotcloud.com/`.

[58] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: a retrospective. In *Proceedings of New Security Paradigms Workshop*, Caledon Hills, Ontario, September 2000.

[59] Peyman Faratin, David Clark, Patrick Gilmore, Steven Bauer, Arthur W. Berger, and William Lehr. Complexity of Internet Interconnections: Technology, Incentives and Implications for Policy. In *Annual Telecommunications Policy Research Conference*, Arlington, Virginia, September 2007.

[60] Nathan Farrington, Erik Rubow, and Amin Vahdat. Data Center Switch Architecture in the Age of Merchant Silicon. In *Proceedings of Hot Interconnects*, New York, New York, August 2009.

[61] Nick Feamster and Hari Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, May 2005.

[62] Wu-chang Feng and Travis Schluessler. The Case for Network Witnesses. In *Proceedings of Workshop on Secure Network Protocols (NPSEC)*, Orlando, Florida, October 2008.

[63] Sally Floyd and Kevin Fall. Promoting the use of end-to-end congestion control in the Internet. *IEEE/ACM Transactions on Networking (TON)*, 7(4):458–472, August 1999.

[64] Pierre Francois, Clarence Filsfils, John Evans, and Olivier Bonaventure. Achieving sub-second IGP convergence in large IP networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 35(3):35–44, July 2005.

[65] Daniel Friedman. The double auction market institution: A survey. *The Double Auction Market: Institutions, Theories, and Evidence*, pages 3–25, 1993.

[66] Morrie Gasser, Andy Goldstein, Charlie Kaufman, and Butler Lampson. The Digital Distributed System Security Architecture. In *Proceedings of NIST National Computer Security Conference*, pages 305–319, Baltimore, Maryland, October 1989.

[67] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google file system. *ACM SIGOPS Operating Systems Review (OSR)*, 37(5):29, December 2003.

[68] Sharon Goldberg, Aaron D. Jaggard, and Rebecca N. Wright. Rationality and Traffic Attraction: Incentives for Honest Path Announcements in BGP. In *Proceedings of ACM SIGCOMM*, Seattle, Washington, August 2008.

[69] Jamaloddin Golestani and Krishan Sabnani. Fundamental observations on multicast congestion control in the Internet. In *Proceedings of IEEE INFOCOM*, New York, New York, March 1999.

[70] Albert Greenberg, James Hamilton, David A. Maltz, and Parveen Patel. The Cost of a Cloud: Research Problems in Data Center Networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 39(1), January 2009.

[71] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. VL2: a scalable and flexible data center network. *ACM SIGCOMM Computer Communication Review (CCR)*, 39(4), October 2009.

[72] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers,

Jennifer Rexford, Geoffrey Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM Computer Communication Review (CCR)*, 35(5):41–54, October 2005.

[73] Tim Greene. SAS 70 is the measure of cloud security. `http://www.networkworld.com/newsletters/2009/062309cloudsec1.html`.

[74] Natasha Gude, Teemu Koponen, Justin Pettit, Ben Pfaff, Martín Casado, Nick McKeown, and Scott Shenker. NOX: Towards an Operating System for Networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(3):105–110, July 2008.

[75] Ajay Gulati and Carl A. Waldspurger. PARDA: Proportional Allocation of Resources for Distributed Storage Access. In *Proceedings of USENIX File and Storage Technologies (FAST)*, San Jose, California, February 2009.

[76] Ramakrishna Gummadi, Hari Balakrishnan, Petros Maniatis, and Sylvia Ratnasamy. Not-a-Bot (NAB): Improving Service Availability in the Face of Botnet Attacks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, April 2009.

[77] Chuanxiong Guo, Guohan Lu, Dan Li, Haitao Wu, Xuan Zhang, Yunfeng Shi, Chen Tian, Yongguang Zhang, and Songwu Lu. BCube: A High Performance, Server-centric Network Architecture for Modular Data Centers. *ACM SIGCOMM Computer Communication Review (CCR)*, 39(4):63–74, October 2009.

[78] Chuanxiong Guo, Guohan Lu, Helen J. Wang, Shuang Yang, Chao Kong, Peng Sun, Wenfei Wu, and Yongguang Zhang. SecondNet: A Data Center Network Virtualization Architecture with Bandwidth Guarantees. In *Proceedings of ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Philadelphia, Pennsylvania, December 2010.

[79] Andreas Haeberlen, Ioannis Avramopoulos, Jennifer Rexford, and Peter Druschel. NetReview: Detecting when interdomain routing goes wrong. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, April 2009.

[80] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum,

and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. In *Proceedings of USENIX Security Symposium*, San Jose, California, July 2008.

[81] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: a GPU-Accelerated Software Router. In *Proceedings of ACM SIGCOMM*, New Delhi, India, August 2010.

[82] James Hendricks and Leendert van Doorn. Secure Bootstrap Is Not Enough: Shoring up the Trusted Computing Base. In *Proceedings of SIGOPS European Workshop*, Leuven, Belgium, August 2004.

[83] Hewlett-Packard. HP ProCurve 2910al Switch Series. `http://h10146.www1.hp.com/products/switches/HP_ProCurve_2910al_Switch_Series/overview.htm/`.

[84] C. Hoff, S. Johnston, G. Reese, and B. Sapiro. CloudAudit 1.0 - Automated Audit, Assertion, Assessment, and Assurance API (A6), January 2010. IETF Internet-Draft.

[85] IBM Corporation. IBM Extends Enhanced Data Security to Consumer Electronics Products. Press Release, April 2006. `http://www-03.ibm.com/press/us/en/pressrelease/19527.wss`.

[86] International Organization for Standardization. Information Processing Systems - Open Systems Interconnection, Management Information Protocol Specification - Part 2: Common Management Information Protocol, December 1988. ISO DIS 9596-2.

[87] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: Fair Scheduling for Distributed Computing Clusters. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, Montana, October 2009.

[88] Giuseppe F. Italiano, Rajeev Rastogi, and Bülent Yener. Restoration Algorithms for Virtual Private Networks in the Hose Model. In *Proceedings of IEEE INFOCOM*, New York, New York, June 2002.

[89] Mahesh Kallahalla, Mustafa Yusal, Ram Swaminathan, David Lowell, Mike Wray, Tom Christian, Nigel Edwards, Chris I. Dalton, and Frederic Gittler. SoftUDC: A Software-Based Data Center for Utility Computing. *Computer*, 37(11):38–46, November 2004.

[90] Srikanth Kandula, Jitu Padhye, and Paramvir Bahl. Flyways to De-congest Data Center Networks. In *Proceedings of ACM HotNets*, New York, New York, October 2009.

[91] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. Nature of Datacenter Traffic: Measurements and Analysis. In *Proceedings of ACM Internet Measurement Conference (IMC)*, Chicago, Illinois, November 2009.

[92] Thomas Karagiannis, Richard Mortier, and Antony Rowstron. Network Exception Handlers: Host-network Control in Enterprise Networks. In *Proceedings of ACM SIGCOMM*, Seattle, Washington, August 2008.

[93] Dina Katabi, Mark Handley, and Charlie Rohrs. Congestion control for high bandwidth-delay product networks. In *Proceedings of ACM SIGCOMM*, Pittsburgh, Pennsylvania, August 2002. ACM Press.

[94] Eric Keller, Ruby B. Lee, and Jennifer Rexford. Accountability in hosted virtual networks. In *Proceedings of ACM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)*, Barcelona, Spain, August 2009.

[95] Stephen Kent, Charles Lynn, Joanne Mikkelson, and Karen Seo. Secure Border Gateway Protocol (S-BGP) – Real World Performance and Deployment Issues. In *Proceedings of ISOC Symposium on Network and Distributed System Security*, San Francisco, CA, February 2000.

[96] Jeremy Kirk. 'Evil twin' hotspots proliferate, April 2007. `http://www.pcworld.com/businesscenter/article/131199/evil_twin_hotspots_proliferate.html`.

[97] Teemu Koponen, Martin Casado, Natasha Gude, Jeremy Stribling, Leon Poutievski, Min Zhu, Rajiv Ramanathan, Yuichiro Iwata, Hiroaki Inoue, Takayuki Hama, and Scott Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Vancouver, British Columbia, October 2010.

[98] Orran Krieger, Phil McGachey, and Arkady Kanevsky. Enabling a marketplace of clouds: VMware's vCloud director. *ACM SIGOPS Operating Systems Review (OSR)*, 44:103–114, December 2010.

[99] Horacio Andrés Lagar-Cavilla, Joseph Andrew Whitney, Adin Matthew Scannell, Philip Patchin, Stephen M. Rumble, Eyal de Lara, Michael Brudno, and Mahadev Satyanarayanan. SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing. *Proceedings of ACM EuroSys*, March 2009.

[100] Terry Lam, Sivasankar Radhakrishnan, Amin Vahdat, and George Varghese. NetShare : Virtualizing Data Center Networks across Services. Technical Report CS2010-0957, University of California, San Diego, May 2010.

[101] Linda Leung. Could A Cloud Computing Exchange Work? `http://www.datacenterknowledge.com/archives/2010/01/19/could-a-cloud-computing-exchange-work/`.

[102] Ang Li, Xiaowei Yang, Srikanth Kandula, and Ming Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of ACM Internet Measurement Conference (IMC)*, Melbourne, Australia, November 2010.

[103] Xin Liu, Xiaowei Yang, David Wetherall, and Thomas Anderson. Efficient and Secure Source Authentication with Packet Passports. In *Proceedings of Workshop on Steps to Reducing Unwanted Traffic on the Internet (SRUTI)*, San Jose, California, July 2006.

[104] Guohan Lu, Chuanxiong Guo, Yulong Li, and Zhiqiang Zhou. ServerSwitch : A Programmable and High Performance Platform for Data Center Networks. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, Massachusetts, March 2011.

[105] Sridhar Machiraju and Randy H. Katz. Reconciling Cooperation with Confidentiality in Multi-Provider Distributed Systems. Technical Report UCB/CSD-4-1345, Computer Science Division (EECS), University of California, Berkeley, 2004.

[106] Harsha V. Madhyastha, Tomas Isdal, Michael Piatek, Colin Dixon, Thomas Anderson, Arvind Krishnamurthy, and Arun Venkataramani. iPlane: An Information Plane for Distributed Services. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, Washington, November 2006.

[107] Daniel W. Manchala. E-Commerce Trust Metrics and Models. *IEEE Internet Computing*, 4(2):36–44, April 2000.

[108] Steve McCanne and Van Jacobson. The BSD packet filter: A new architecture for user-level packet capture. In *Proceedings of USENIX Winter Conference*, San Diego, California, January 1993.

[109] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review (CCR)*, 38(2), 2008.

[110] Microsoft. An Overview of Windows Azure. `http://download.microsoft.com/download/A/A/6/AA6A260A-B920-4BBC-AE33-8815996CD8FB/02-ArticleIntroductiontoWindowsAzure.docx`.

[111] Jayaram Mudigonda, Praveen Yalagandula, Mohammed Al-Fares, and Jeffrey C. Mogul. SPAIN: COTS Data-Center Ethernet for Multipathing over Arbitrary Topologies. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Jose, California, April 2010.

[112] Steven Nagy. The Azure Fabric Controller. `http://azure.snagy.name/blog/?p=89`.

[113] Jad Naous, Ryan Stutsman, David Mazieres, Nick McKeown, and Nickolai Zeldovich. Delegating Network Security Through More Information. In *Proceedings of ACM Workshop on Research on Enterprise Networking*, Barcelona, Spain, August 2009.

[114] NASA. Openstack Nova. `http://nova.openstack.org/`.

[115] Priscilla Oppenheimer. *Top-Down Network Design*. Cisco Press, 2004.

[116] Jeff Z. Pan. Resource Description Framework. In *Handbook on Ontologies*, pages 71–90. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[117] Rong Pan, Balaji Prabhakar, and Ashvin Laxmikantha. QCN: Quantized Congestion Notification. `http://www.ieee802.org/1/files/public/docs2007/au-prabhakar-qcn-description.pdf`, May 2007.

[118] Jeffrey Pang, Ben Greenstein, Michael Kaminsky, and Damon Mccoy. Improving Wireless Network Selection with Collaboration. In *Proceedings*

of ACM International Conference on Mobile Systems, Applications, and Services (MobiSys), Krakow, Poland, June 2009.

[119] Bryan Parno, Zongwei Zhou, and Adrian Perrig. Don't Talk to Zombies: Mitigating DDoS Attacks via Attestation. Technical Report CMU-CyLab-09-009, CyLab, Carnegie Mellon University, June 2009.

[120] Bryan Parno, Zongwei Zhou, and Adrian Perrig. Help Me Help You: Using Trustworthy Host-Based Information in the Network. Technical Report CMU-CyLab-09-016, CyLab, Carnegie Mellon, November 2009.

[121] Radia Perlman. *Network layer protocols with Byzantine robustness*. PhD thesis, Massachusetts Institute of Technology, August 1988.

[122] Peter Phaal and Marc Lavine. sFlow Version 5. `http://www.sflow.org/sflow_version_5.txt`, July 2004.

[123] Lucian Popa, Steven Y. Ko, and Sylvia Ratnasamy. CloudPolice: Taking Access Control out of the Network. In *Proceedings of ACM HotNets*, Monterey, California, October 2010.

[124] Barath Raghavan, Kashi Vishwanath, Sriram Ramabhadran, Kenneth Yocum, and Alex C. Snoeren. Cloud control with distributed rate limiting. In *Proceedings of ACM SIGCOMM*, Kyoto, Japan, August 2007.

[125] Anirudh Ramachandran, Kaushik Bhandankar, Mukarram Bin Tariq, and Nick Feamster. Packets with Provenance. Technical Report GT-CS-08-02, Georgia Institute of Technology, 2008.

[126] Patrick Reynolds, Oliver Kennedy, Emin Gün Sirer, and Fred B. Schneider. Securing BGP using external security monitors. Technical Report TR2006-2065, Computer Science Department, Cornell University, December 2006.

[127] Injong Rhee and Lisong Xu. CUBIC: A New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review (OSR)*, 42(5):64–74, July 2008.

[128] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proceedings of ACM Conference on*

Computer and Communications Security (CCS), Chicago, Illinois, November 2009.

[129] Eric Rosen, Arun Viswanathan, and Ross Callon. Multiprotocol Label Switching Architecture, January 2001. IETF RFC 3031.

[130] Guy Rosen. Anatomy of an Amazon EC2 Resource ID. `http://www.jackofallclouds.com/2009/09/anatomy-of-an-amazon-ec2-resource-id/`.

[131] Reiner Sailer, Trent Jaeger, Xiaolan Zhang, and Leendert van Doorn. Attestation-based policy enforcement for remote access. In *Proceedings of ACM Conference on Computer and Communications Security (CCS)*, Washington, District of Columbia, October 2004.

[132] Jose Renato Santos, Yoshio Turner, G. John Janakiraman, and Ian Pratt. Bridging the Gap between Software and Hardware Techniques for I/O Virtualization. Technical Report HPL-2008-39, HP Labs, 2008.

[133] Fred B. Schneider, Kevin Walsh, and Emin Gün Sirer. Nexus Authorization Logic (NAL): Design Rationale and Applications. *ACM Transactions on Information and System Security*, 14(1):8–35, September 2010.

[134] Vyas Sekar, Michael K. Reiter, Walter Willinger, Hui Zhang, Ramana Rao Kompella, and David G. Andersen. CSAMP: A System for Network-Wide Flow Monitoring. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, April 2008.

[135] Elaine Shi, Adrian Perrig, and Leendert Van Doorn. BIND: A Fine-Grained Attestation Service for Secure Distributed Systems. In *Proceedings of IEEE Symposium on Security and Privacy*, Oakland, California, May 2005.

[136] Alan Shieh, Srikanth Kandula, Albert Greenberg, and Changhoon Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *Proceedings of USENIX HotCloud*, Boston, Massachusetts, June 2010.

[137] Alan Shieh, Andrew C. Myers, and Emin Gün Sirer. A Stateless Approach to Connection-oriented Protocols. *ACM Transactions on Computer Systems (TOCS)*, 26(3):1–50, September 2008.

[138] Manpreet Singh, Prashant Pradhan, and Paul Francis. MPAT: Aggregate

TCP Congestion Management as a Building Block for Internet QoS. In *Proceedings of IEEE International Conference on Network Protocols (ICNP)*, Berlin, Germany, October 2004.

[139] Emin Gün Sirer, Willem de Brujin, Patrick Reynolds, Alan Shieh, Kevin Walsh, Dan Williams, and Fred B. Schneider. Logical Attestation: An Authorization Architecture for Trustworthy Computing. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Cascais, Portugal, October 2011.

[140] Ning So and Hao-Hsin Huang. Building a Highly Adaptive, Resilient, and Scalable MPLS Backbone. In *Proceedings of MPLS World Congress*, Paris, France, February 2007.

[141] Paolo Victor Soares, Jose Renato Santos, Niraj Tolia, Dorgival Guedes, and Yoshio Turner. Gatekeeper: Distributed Rate Control for Virtualized Datacenters. Technical Report HPL-2010-151, HP Labs, October 2010.

[142] Lakshminarayanan Subramanian, Volker Roth, Ion Stoica, Scott Shenker, and Randy H. Katz. Listen and Whisper: Security Mechanisms for BGP. In *Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, San Francisco, California, March 2004.

[143] G. Edward Suh, Dwaine Clarke, Blaise Gasend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of IEEE MICRO*, San Diego, California, December 2003.

[144] Sunay Tripathi, Nicolas Droux, Thirumalai Srinivasan, and Kais Belgaied. Crossbow: from hardware virtualized NICs to virtualized networks. In *Proceedings of ACM Workshop on Virtualized Infrastructure Systems and Architectures (VISA)*, Barcelona, Spain, August 2009.

[145] Trusted Computing Group. TCG Trusted Network Connect: TNC Architecture for Interoperability, Specification Version 1.3, April 2008.

[146] Vytautas Valancius, Nick Feamster, Ramesh Johari, and Vijay Vazirani. MINT: A Market for INternet Transit. In *Proceedings of Workshop on Re-Architecting the Internet (ReArch)*, Madrid, Spain, December 2008.

[147] Luis M. Vaquero, Luis Rodero-merino, Juan Caceres, and Maik Lindner. A Break in the Clouds : Towards a Cloud Definition. *ACM SIGCOMM Computer Communication Review (CCR)*, 39(1):50–55, January 2009.

[148] Ymir Vigfusson, Hussam Abu-Libdeh, Mahesh Balakrishnan, Ken Birman, and Yoav Tock. Dr. Multicast: Rx for Data Center Communication Scalability. In *Proceedings of Workshop on Large-scale Distributed Systems and Middleware (LADIS)*, Yorktown, New York, September 2008. ACM Press.

[149] VMware Inc. VMWARE-VMINFO-MIB. `http://www.oidview.com/mibs/6876/VMWARE-VMINFO-MIB.html`.

[150] Carl A. Waldspurger. *Lottery and Stride Scheduling : Proportional Share Resource Management*. PhD thesis, MIT, September 1995.

[151] Michael Walfish, Jeremy Stribling, and Maxwell Krohn. Middleboxes no longer considered harmful. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Francisco, California, December 2004.

[152] Guohui Wang and T.S. Eugene Ng. The Impact of Virtualization on Network Performance of Amazon EC2 Data Center. In *Proceedings of IEEE INFOCOM*, San Diego, California, March 2010.

[153] Joe Weinman. Hedging Your Options for the Cloud. `http://gigaom.com/2009/12/13/hedging-your-options-for-the-cloud/`, December 2009.

[154] Bill Woodcock and Vijay Adhikari. Survey of Characteristics of Internet Carrier Interconnection Agreements. Technical report, Packet Clearing House, May 2011.

[155] Haitao Wu, Zhenqian Feng, Chuanxiong Guo, and Yongguang Zhang. ICTCP: Incast Congestion Control for TCP in Data Center Networks. In *Proceedings of ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Philadelphia, Pennsylvania, December 2010.

[156] Andrew Chi-Chih Yao. Protocols for Secure Computations (Extended Abstract). In *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS)*, Chicago, Illinois, November 1982.

[157] Matei Zaharia, Andy Konwinski, Anthony D. Joseph, Randy Katz, and Ion Stoica. Improving MapReduce Performance in Heterogeneous Environments. In *Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, San Diego, California, December 2008.

[158] Xin Zhang, Abhishek Jain, and Adrian Perrig. Packet-dropping Adversary Identification for Data Plane Security. In *Proceedings of ACM International Conference on Emerging Networking EXperiments and Technologies (CoNEXT)*, Madrid, Spain, December 2008.

[159] Wenchao Zhou, Yun Mao, Boon Thau Loo, and Martín Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *Proceedings of IEEE International Conference on Data Engineering (ICDE)*, Shanghai, China, April 2009.