Packet Classification as a Fundamental Network Primitive

Dilip Antony Joseph

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-63
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-63.html

May 15, 2009

Packet Classification as a Fundamental Network Primitive

by

Dilip Antony Joseph

B.Tech. (Indian Institute of Technology, Madras) 2004
M.S. (University of California, Berkeley) 2006

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

Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor Ion Stoica, Chair
Professor Scott Shenker
Professor John Chuang

Spring 2009

The dissertation of Dilip Antony Joseph is approved.

Chair	Date
	Date
	Date

University of California, Berkeley

Spring 2009

Packet Classification as a Fundamental Network Primitive

by

Dilip Antony Joseph

Abstract

Packet Classification as a Fundamental Network Primitive

by

Dilip Antony Joseph

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor Ion Stoica, Chair

Packet classification is an ubiquitous and key building block of many critical network functions like routing, firewalling, and load balancing. However, classification is currently implemented, deployed and configured in an ad-hoc manner. Reliance on ad-hoc mechanisms make classification hard to configure, inefficient and inflexible.

In this thesis, we address the above limitations by elevating packet classification as a fundamental network primitive. We do so by introducing a new *classification layer* in the network protocol stack, and by defining two control plane protocols – policy-based classifier deployment and generic classification offload. In policy-based classifier deployment, packets are explicitly redirected through the classifiers specified by network policy. Generic classification offload provides a signaling mechanism that enables different entities to collaboratively implement classification. Through prototype implementations, testbed experiments and formal analysis, we demonstrate that our solution simplifies deployment and configuration, and improves flexibility, efficiency and performance of packet classification.

Professor Ion Stoica
Dissertation Committee Chair

To my parents

Contents

List of Figures

List of Tables

Acknowledgements

I am very fortunate to have Ion Stoica as my adviser. Ion has been a constant source of support and inspiration in my Ph.D. career. The ideas presented in this thesis originated from and were nurtured through many discussions with Ion. He not only helped in architecting the high level research ideas, but also often deeply dived into C++ code, trying to scrape out that last bit of performance. I am indebted to Ion for his guidance and encouragement.

My thesis committee members John Chuang and Scott Shenker have, from time to time, infused fresh perspectives and suggested new directions for my thesis research. I am grateful to them for their help and guidance. I also greatly enjoyed working with John on our paper on modeling the adoption of new Internet architectures.

I thank Randy Katz for serving on my qualifying exam committee and providing valuable feedback. By introducing me to his contacts at various networking companies, Randy helped me present my work to the industry. It was Randy who recommended the IEEE Networks Special Issue on Middleboxes as an apt venue to publish my middlebox modeling work.

I greatly enjoyed working with Vern Paxson as a Teaching Assistant for EE122. Vern was an amazing mentor. I was always inspired by his extraordinary dedication and attention to detail.

My internship at Microsoft Research provided me a wonderful opportunity to experience research at an industry lab. I thank my mentors Venkata Padmanabhan and Sharad Agarwal for making my internship a very enjoyable and rewarding research experience.

I am grateful to many colleagues for supporting my research publications, both as collaborators and reviewers. Ayumu Kubota, Karthik Lakshminarayanan, Jayanth Kannan and Klaus Wehrle were instrumental in designing and writing up OCALA. Nikhil Shetty played a big role in designing and refining the models in the paper on new Internet architecture adoption. Arsalan Tavakoli played a key role in designing the policy-aware switching layer. The feedback provided by Lisa Fowler, Ganesh Ananthanarayanan, Lucian Popa, Gautam Altekar, Jayanth Kannan, Andrey Ermolinskiy, Daekyeong Moon, Yanpei Chen,

Rodrigo Fonseca, Byung-Gon Chun, Brighten Godfrey and Mythili Vutukuru has been very useful in refining my research and publications.

Most of the empirical results in this thesis were obtained from experiments performed on the DETER testbed. Experiments for research associated with my M.S. degree were performed on PlanetLab. I am grateful to the creators of DETER and PlanetLab for offering such fine experimentation platforms. I thank Anthony Joseph for providing me access to DETER.

I thank Keith Sklower, Mike Howard and Jon Kuroda for their prompt technical support and recommendations on my experimental setup. Keith graciously spent a lot of time helping me debug a complex policy-aware switching layer forwarding problem. It was Mike who first recommended using DETER for my experiments. The cheerful manner with which Jon responded to support requests always lightened up my day.

Working at Will Sobel's and Athulan Vijayaraghavan's startup was a very fun and rewarding experience. I learnt more about real-world programming from Will than from anyone else.

Damon Hinson, Kattt Atchley and La Shana Porlaris always happily helped me deal with administrative issues. The staff at the Berkeley International Office were very helpful in addressing problems due to the discrepancy in my name ordering. I am grateful to all of them for making my life at UC Berkeley smooth.

I had the good fortune of being in the office cubicle with probably the maximum occupancy rate. Arsalan, Daekyeong, Andrey, and Gautam (before RADLab was built) were in office all the time. I greatly enjoyed the many research and non-research discussions we have had. Arsalan was always generous with his M&Ms and jokes about me.

My stay at Berkeley would not have been the same without squash. I am very grateful to my coach Ian Hicks for his encouragement. I had a very enjoyable time playing with Athulan Vijayaraghavan, Steven Go, Mohammed Shamma, Charlie Browning, Andrew Green, Robert Ashmore, Nick Paget, and Lewis.

My friends - Adarsh Krishnamurthy, Athulan Vijayaraghavan, Archana Sekhar, David

Chu, Jayanth Kannan, Karthik Lakshminarayanan, Kranthi Kiran, Lucian Popa, Lakshmi Subramanian, Praveena Garimelli, Sameer Vermani, Sriram Sankararaman, Subramaniam Venkatraman, and Varun Kacholia made life fun and enjoyable. I am very grateful for their company and the times we spent together.

My parents, Dr. P.S. Joseph and Mrs. Thresiamma Joseph, and my brother Deepu were the main inspiring factors behind my decision to pursue a Ph.D. They, along with the rest of my family, have been the pillars of support throughout my Ph.D. journey. I thank my parents, my brother and my sister-in-law Neena, my parents-in-law Mr. V.J. Mathew and Mrs. Annie Mathew, my sister-in-law Elizabeth and her husband Bijoy, my nephew Mathew and my niece Neha, and my brother-in-law Joseph for their encouragement and prayers.

My wife Mary has always been a source of love and strength, sharing with me the ups (read: paper getting accepted) and downs (read: paper getting rejected) of Ph.D. life. Her loving support, enthusiasm for travel and fabulous cooking made my life fun and comfortable. This journey would not have been enjoyable or meaningful without her.

Finally, I am grateful to God Almighty for His many blessings, including a smooth Ph.D. journey.

Chapter 1

Introduction

Packet classification is a key building block of many critical network functions. Every packet in a network encounters classification at one or more entities. In Figure 1.1, *forwarding elements* like layer-2 switches and layer-3 routers, as well as special-purpose classifiers like a firewall and a load balancer, classify a packet as they forward it from endhost A to web server B. A router classifies the packet to determine where it should be forwarded to and what QoS it should receive. A load balancer classifies the packet to identify the web server instance to which it must be forwarded. A firewall classifies the packet based on its security policies to decide whether to drop the packet or not.

Classification in today's networks faces three main limitations – configuration hardness, inflexibility, and inefficiency. These limitations are results of the following four characteristics of packet classification: (i) *Complexity* of classification operations, (ii) *Topology Mismatch* between network administrators and forwarding mechanisms, (iii) *Semantic Gap* between entities on the packet path, and (iv) *Resource Mismatch* between entities on the packet path.

Complexity : Classification remains a complex operation, in spite of significant improvements in classification speed over the last two decades [63, 82, 91]. It creates performance and scalability bottlenecks, especially in the face of rapidly swelling traffic volumes and

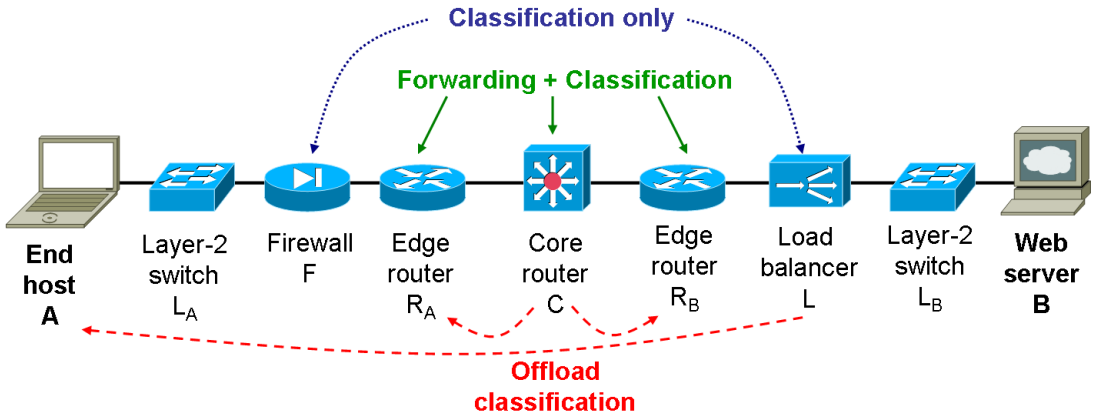

Figure 1.1. Packet classification is ubiquitous in networks.

bandwidth. Classification involving multiple packet header fields exhibits a fundamental trade-off between computational complexity and memory requirements [63]. Classification involving deep packet inspection and application payload parsing is even more complex. In a benchmark of commercial application delivery controllers [12] (*i.e.,* enhanced load balancers), classification operations involving basic layer-7 HTTP processing attained up to 78% fewer connections/second and up to 30% lower throughput than those that used simple layer-4 processing.

Topology Mismatch : A network administrator requires different types of traffic to be processed by different sequences of classifiers in order to satisfy application needs and network policy. These classifier sequences constitute the desired logical network topology. In today's networks, there is a mismatch between the desired logical topology and classifier deployment mechanisms. Classifiers are currently deployed by placing them on the network path taken by packets. A packet's path thus *implicitly* determines the sequence of classifiers that process it. In Figure 1.1, a packet from A to B is processed by the firewall F and the load balancer L because they lie on its path. The path taken by a packet is jointly determined by a wide variety of interacting factors – physical network topology, layer-3 routing protocols like BGP [1] and OSPF [31], layer-2 spanning tree construction [10], VLAN [16] configuration and traffic engineering [32]. These factors are unaware of the logical topology intended by the network administrator.

Semantic Gap : Different entities on a packet's path, including the source and destination, have different amounts of semantic and local context related to a particular classification application. For instance, a web load balancer must forward all packets in an HTTP 'session' to the same web server instance for semantic correctness or improved performance through caching. The web browser at a client has more local semantic context to identify the multiple HTTP connections in a user's HTTP session than the load balancer. Similarly, an edge router is in a better position to determine a packet's QoS than a core router. The edge router's closeness to traffic sources provides access to finer-grained local QoS policies and higher precision traffic accounting than a far away core router.

Resource Mismatch : Similar to the semantic gap, different entities on a packet's path have different amounts of processing resources (*e.g.,* CPU, memory) to spend on classifying each packet. For example, a core router can process more packets per second than an edge router. However, the substantially higher traffic load at a core router allows it lesser processing time per packet than an edge router. Thus, there is a resource mismatch between an edge router and a core router.

Classifiers use *classification offload* to bridge the semantic gap in placement of classification operations, and to leverage the resource mismatch in overcoming performance and scalability bottlenecks caused by classification complexity. Classifiers *push* classification tasks to entities with more semantic context or more per-flow/packet processing resources. For instance, a load balancer bridges the semantic gap by pushing HTTP session identification to endhosts using HTTP cookies [15]. An endhost uses local context to identify the various HTTP connections in a particular HTTP session. It embeds the same 'session' cookie value in all connections in the session. The load balancer uses the cookie value in an HTTP connection to identify its session, and thereby select the correct webserver instance.

Although many solutions have been proposed for classification offload, they are point, and often ad-hoc, solutions that work only in the context of a single layer (*e.g.,* network, transport, or application) and a single application (*e.g.,* QoS, HTTP load balancing). For example, unlike a layer-7 load balancer, a core router uses MPLS [25] at layer 2.5 to offload route lookup to less-loaded upstream edge routers. An edge router performs expensive

route lookup operations on behalf of the core router and conveys the lookup results to it by tagging packets with different labels.

1.1 Limitations of Current Classification Solutions

Implicit on-path classifier deployment, ad-hoc classification offload solutions, and the non-availability of clear information about classifier behavior make classification in today's networks hard to configure, inflexible and inefficient. We briefly describe these limitations in this section.

1.1.1 Configuration Hardness

The sequence of classifiers traversed by a packet implicitly depends on the network path it takes. This makes deployment and configuration of classifiers complex and error-prone. A classifier deployment requires careful and coordinated configuration of all factors that affect network paths – cabling, spanning tree link weights, IP routing, VLAN configuration, traffic engineering, and failover protocols. Complex and often unpredictable interactions between these factors makes it challenging to guarantee that a packet traverses all the required classifiers in the correct sequence under all network scenarios. For instance, a crucial classifier like a firewall may be bypassed when network paths change as a result of link/router failure, new link addition, traffic engineering, or activation of a new ingress point triggered by Internet routing changes.

The lack of a generic classification offload solution results in the proliferation of different, often ad-hoc, mechanisms for different applications at different protocol layers : for example, MPLS at layer-2.5 for offloading route lookup, and HTTP cookies at layer-7 for HTTP session identification. A network administrator must carefully implement and configure all these different mechanisms for correct network operation. This increases network configuration complexity and the probability for error.

Current classification offload solutions do not enable explicit coordination between dif-

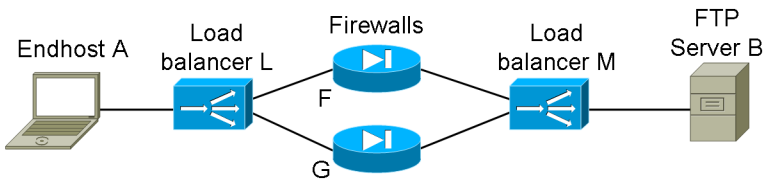

Figure 1.2. Coordinating multiple *classifiers* for firewall load balancing.

ferent entities involved in packet classification. This further increases configuration and operational complexity. Consider the pair of firewall load balancers, L and M, in Figure 1.2. The classification decision at a firewall instance (say, F) uses stored information about packets previously seen (if any) in the opposite flow direction. Hence, the load balancers must select the same firewall instance for both forward and reverse directions of a TCP flow. They currently coordinate their decisions implicitly based on the physical wiring configuration and by remembering the 5-tuples of packets that arrived on each network interface [74]. Correctly wiring the various entities and configuring such coordination are complex and error-prone tasks.

The non-availability of clear information about the varied configuration requirements and operational behavior of different classifiers further complicates network configuration. For instance, a load balancer often rewrites some of the fields in packets received by it. This rewriting must be taken into consideration while configuring the filtering rules at a downstream firewall. Detailed information about the packet transformations performed by classifiers is often buried deep inside technical manuals or simply absent. Network deployment and configuration complexity is aggravated by the lack of easy access to such information.

1.1.2 Inflexibility

Evolving network functionality and applications require changes in the logical network topology, *i.e.,* the sequence of classifiers which process different types of traffic. For instance, increased security threats may necessitate the addition of an intrusion detection box (IDS)

that scans all existing network traffic. A new application may require that a new load balancer be deployed, and that all traffic of the application traverse it.

Current classifier deployment mechanisms tightly couple a network's logical topology with its physical topology. This makes the network inflexible and restricts evolvability. For instance, adding an IDS often requires identifying a choke point through which all traffic flows and physical re-cabling to insert the IDS there. Alternatively, the IDS is placed at a non-choke point, and forwarding configuration parameters and VLAN settings are tweaked to force all traffic through it. Such tweaking interacts with other network factors like traffic engineering, fault tolerance and traffic isolation, often in unpredictable ways. Hence, indirect mechanisms to change the logical network topology are complex, error-prone and hard to deploy.

Current classification offload mechanisms are also inflexible as they are tailored to specific applications and specific layers. Many existing classification applications like firewalling are deprived of the benefits of classification offload. A future classification application may require designing and implementing yet another offload mechanism.

1.1.3 Inefficiency

Current classifier deployment solutions gratuitously spend classifier resources by forcing classifiers to process irrelevant traffic or by preventing them from processing relevant traffic. All traffic on a particular network path is forced to traverse the same sequence of classifiers, irrespective of application requirements. For instance, a web load balancer wastes valuable processing resources if it must unnecessarily process non-HTTP traffic flowing on the network path on which it is deployed. On the other hand, classifier resources are also wasted if they are deployed on an inactive network path. For instance, even if the load balancer on the active path is reeling under heavy load, the equally capable (and expensive!) load balancer on the backup path remains unutilized.

Current classification offload solutions also waste classifier resources by relying on expensive packet processing. To read the session information in the HTTP cookie [15] embedded

by an endhost, a load balancer reconstructs the transport stream, performs SSL decryption (if encrypted) and parses HTTP headers. Such heavy-weight layer-7 processing can incur high performance penalties – up to 78% reduction in connections/second and up to 30% reduction in throughput, as per the benchmark mentioned earlier in this chapter. This expensive packet processing is not a critical component of classification offload, but is simply an inefficiency of current mechanisms.

The limited scope of existing classification offload mechanisms deprives many applications of the benefits of classification offload. In addition to inflexibility, this leads to inefficiency. Many applications experience inefficiencies that could have been alleviated by classification offload. For instance, correct firewall operation in Figure 1.2 requires that the control and data connections in an FTP session be processed by the same firewall instance. Currently, the firewall load balancers implement FTP protocol analysis on the received packets to identify the control and data connections. If an offload mechanism for firewall load balancing existed, they could have simply relied on hints from the endhosts to perform such classification.

1.2 Our Contributions

This thesis attempts to address the limitations of current classification solutions described in the previous section, *i.e.,* simplify classifier deployment and configuration, and improve efficiency and flexibility. We argue that the root cause for these limitations is that packet classification is treated in an ad-hoc manner, in spite of its ubiquity in today's networks.

The main contribution of this thesis is *to elevate packet classification as a fundamental primitive in the protocol stack.* We do so by defining a new *classification layer*, or *CLayer*, in the protocol stack, and a pair of associated control plane mechanisms – *policy-based classifier deployment* and *generic classification offload.* Together with these mechanisms, the *CLayer* provides a holistic approach to overcome the topology, semantic and resource

mismatches inherent in today's classification solutions, thereby simplifying configuration, and improving efficiency and flexibility.

In a nutshell, our proposal works as follow: The *CLayer* in a packet's header carries information used by forwarding elements and classifiers to efficiently classify packets or to forward them to the required classifiers. In *policy-based classifier deployment*, forwarding elements *explicitly* redirect different types of traffic through the sequence of classifiers specified by administrative policy. Once the desired logical classifier sequence has been enforced on the data path, *generic classification offload* configures on-path entities to efficiently and collaboratively implement the various classification operations required on the path. Our proposal is agnostic of the underlying classification operations, and benefits from the large body of research aimed at speeding up individual classification tasks like longest prefix matching [91].

Policy-based classifier deployment and generic classification offload form the two main components of this thesis' contribution. We have designed and prototyped the *Policy-aware Switching Layer* [67], or *PLayer*, as an instantiation of policy-based classifier deployment. We have also designed and prototyped *GOff* as an instantiation of a classification offload protocol. The design, implementation and deployment of classifier-related mechanisms like *PLayer* and *GOff* require a clear understanding about the diverse configuration and operational modes of classifiers. As a third component of this thesis' contribution, we have developed a *model* [66] to understand how different classifiers operate.

Next, we briefly describe these three components of our thesis' contributions, and highlight their benefits.

1.2.1 *PLayer* - Policy-aware Switching Layer

The *PLayer* is a new layer-2 that explicitly supports classifiers and is tailored for data center and enterprise networks. Network administrators specify the sequence of classifiers that must process different types of traffic by configuring 'policies' at a centralized controller. For instance, a policy may specify that all HTTP traffic entering the data center must be

processed by a firewall and then by a load balancer. The controller disseminates these policies to the individual policy-aware switches, or *pswitches*, that are inter-connected to form the *PLayer*. On receiving a frame, a *pswitch* looks up the policy associated with the frame, identifies the classifiers already traversed by it and explicitly forwards it to the next classifier specified by policy. Our *PLayer* design and prototype implementation require no changes to classifiers and endhosts, and only minor changes to network switches.

Centralized policy specification and explicit redirection employed by the *PLayer* bridges the topology mismatch between administrator intent regarding the sequence of classifiers to be traversed by different traffic types and their actual network implementation. Network configuration is simplified as an administrator no longer needs to rely on ad-hoc and intricate configuration hacks to deploy classifiers in the desired sequence. Explicit redirection of packets to classifiers guarantees correct classifier traversal under all network churn conditions. A crucial firewall is never bypassed. The *PLayer* enables efficient utilization of classifiers since packets are redirected only through classifiers specified by policy, and no classifier is stuck unutilized on an inactive network path. *PLayer* enhances data center network flexibility by decoupling the physical and logical topologies. Changing the logical topology in the *PLayer* simply requires a policy change.

1.2.2 *GOff* - Generic Offload Protocol

GOff is a generic signaling protocol that leverages the *CLayer* to simultaneously support offload for multiple classification applications. In *GOff* terminology, entities like firewalls and load balancers which classify a packet and act based on the results of classification are called *classifiers*. Entities like endhosts that aid *classifiers* by performing classification tasks on their behalf are called *helpers*. Through *GOff*, *helpers* advertise their capabilities and *classifiers* notify *helpers* about the classification support they desire. A *helper* on a packet's path performs the requested classification and embeds the results of classification in the packet's easily accessible *CLayer* header. A downstream *classifier* uses these results in speeding up its classification activity.

GOff bridges the semantic gap between the various entities on a packet's path. For instance, the firewall load balancers in Figure 1.2 can rely on endhosts to identify the control and data connections in an FTP session, rather than implementing complex FTP analysis operations themselves. *GOff* also bridges the resource mismatch. For example, *GOff* enables core routers to offload expensive route lookup operations to edge routers, as in MPLS.

GOff avoids the deep-packet inspection inefficiencies of current offload mechanisms by using the *CLayer*. This results in significant performance gains. A *GOff*-enabled layer-4 load balancer achieved 60% more connections/second and throughput than a regular layer-7 load balancer, while supporting similar session semantics. Furthermore, avoiding deep packet inspection simplifies the internal implementation of classifier boxes and lowers cost.

GOff is generic and flexible enough to simultaneously support a variety of classification applications. Moreover, *GOff* enables traditionally centralized classification applications like firewalling to easily and securely offload classification to endhosts, and subsequently improve performance. In our experiments, a firewall implementing *GOff* scalably sustained twice the throughput of a regular firewall as the rule set size increased.

GOff reduces overall network configuration complexity. An administrator no longer needs to separately implement and configure MPLS, HTTP cookies, Diffserv [2], and other point solutions for classification offload. The explicit coordination between classifiers enabled by *GOff* further reduces network configuration complexity. For instance, the firewall load balancers in Figure 1.2 explicitly coordinate with each other using *GOff*, rather than implicitly through ad-hoc mechanisms.

1.2.3 Classifier Model

Our classifier model provides a standard language to succinctly express classifier functionality, configuration and operations. Through sets of *pre-conditions* and *processing rules*, the model describes the types of packets expected by a *classifier* and how it transforms them. We have developed a tool that partially auto-infers a classifier's model through blackbox

testing. Another tool validates the operations performed by a deployed classifier against its pre-specified model.

The succinct and clear description of classifiers helps network administrators plan new network topologies, as well as monitor and debug deployed networks. The model also guides networking researchers. It helps them better understand how their research proposals can work with or fail to work with the wide variety of existing classifiers.

1.3 Thesis Organization

The next chapter provides background information on packet classification – how classifiers are deployed and how classification offload is configured today. We also illustrate the limitations of current approaches in detail.

In Chapter 3, we present a model to represent classifier operations. We illustrate the model using multiple commonly used classifier boxes, and develop semi-automated model inference and validation tools.

In Chapter 4, we introduce the classification layer. We briefly describe its semantics and structure, and provide an overview of the associated protocols – policy-based classifier deployment and generic classification offload.

Chapter 5 describes the design and implementation of the *PLayer*. We demonstrate the *PLayer*'s benefits using experiments on a small testbed, and compare it to related work.

Chapter 6 describes the design and implementation of *GOff*. Through qualitative and quantitative experiments, we demonstrate how *GOff* improves the performance, efficiency, and flexibility of classification. We also describe how *GOff* differs from related efforts aimed at simplifying classification offload.

We summarize our contributions and present future research directions in Chapter 7.

Chapter 2

Background

A *classifier* is any network element that performs classification as part of its packet processing operations. Classifiers can be broadly categorized into two types – *forwarding* classifiers and *special-purpose* classifiers. Forwarding classifiers, as suggested by their name, are classifiers whose main functionality is to forward packets to their destination. For example, a layer-3 router classifies a packet based on its destination IP address to determine the next hop to its final destination. Special-purpose classifiers, on the other hand, are classifiers whose main functionality is not plain packet forwarding. Instead, they implement special-purpose functionality like firewalling and load balancing.

Forwarding classifiers constitute the backbone of networks and determine the network path taken by packets. Special-purpose classifiers are deployed by placing them on these network paths using a variety of mechanisms such as physical wiring, VLANs and tweaking layer-2 and layer-3 forwarding mechanisms. These mechanisms are inefficient, inflexible, and hard to configure. Moreover, they do not guarantee that packets traverse classifiers in the correct sequence under all network conditions. For instance, a critical firewall may be bypassed on adding a new network link.

After the deployment mechanisms determine the sequence of classifiers, the next issue is the *placement* of classification operations. Does a classifier fully implement classification

operations all by itself, or does it share the classification load with other entities in the network? Classifiers may implement classification *in-place*, or may *offload* it to other entities in the network that have more semantic context or per-packet processing resources. Classifiers like firewalls traditionally implement the entire classification operations *in-place*. On the other hand, a load balancer pushes HTTP session identification to endhosts as they have more semantic context to identify HTTP sessions.

The complexity of classification operations and increasing traffic load limits the scalability of in-place classification. Classification offload can alleviate such scalability bottlenecks. Many mechanisms such as HTTP cookies [15], MPLS [25] and Diffserv [2] exist for classification offload. However, the ad-hoc nature and narrow scope of these mechanisms makes them inefficient, inflexible and hard to configure.

Different classifiers process packets differently, and have varied deployment and configuration requirements. Network designers and administrators often have little knowledge about what a specific classifier requires and how it behaves for different traffic types. This lack of knowledge hinders the planning and deployment of new networks, and the monitoring and troubleshooting of existing ones. In addition, the lack of knowledge also hinders networking research activities, especially those concerned with classifiers.

In the rest of this chapter, we first describe current classifier deployment mechanisms in detail. We illustrate their limitations using multiple examples. Secondly, we describe the limitations of in-place classification. Thirdly, we describe current classification offload solutions and their limitations. Fourthly, we explain the problems resulting from the lack of clear knowledge about classifiers.

Before moving on to the description of current classifier deployment mechanisms, we list what we do not cover in this chapter. Firstly, we do not discuss how existing deployment and offload mechanisms compare with the approach proposed by this thesis. Secondly, we do not compare our approach with new proposals like Internet Indirection Infrastructure [88], Delegation Oriented Architecture [94], and Ethane [51]. We defer such comparisons to the Related Work sections of the chapters explaining our approach (Chapters 3, 5 and

6). Thirdly, we do not discuss the large body of research targeted at speeding up low-level classification operations like longest prefix matching. Our approach is agnostic of the algorithms and data structures underlying individual classification operations. For more information about such work, please refer [91].

2.1 Classifier Deployment

In current networks, special-purpose classifiers are *implicitly* deployed by placing them on the network path traversed by packets. Forwarding classifiers like routers and switches form the backbone of the network and are naturally on the packet paths. Special-purpose classifiers like firewalls and load balancers are deployed by physically wiring them on the network paths taken by the relevant traffic or by overloading existing path selection mechanisms to forward traffic through paths containing them.

In this section, we explain in detail how special-purpose classifiers are deployed within a local area or enterprise network. We use a typical data center network topology as an example. Wide area classifier deployments use mechanisms similar to local area deployments. In most cases, wide area deployments are simply formed by smaller local area deployments that use the mechanisms we describe here.

We first describe the typical data center network topology. We then explain how classifiers are deployed and point out their limitations.

2.1.1 Data Center Network Topology

The physical network topology in a data center is typically organized as a three layer hierarchy [44], as shown in Figure 2.1:

1. **Access Layer** provides physical connectivity to the servers in the data centers. It is implemented at the data link layer (*i.e.,* layer-2), as clustering, failover and virtual server movement protocols deployed in data centers require layer-2 adjacency [3, 46].

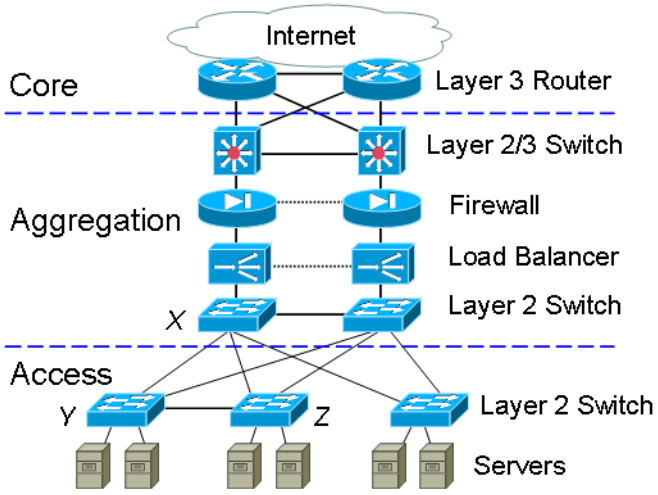

Figure 2.1. Prevalent 3-layer data center network topology.

2. **Aggregation Layer** connects together access layer switches. It commonly operates at both layers 2 and 3. Classifiers like firewalls and load balancers are usually deployed at the aggregation layer to ensure that traffic traverses them before reaching data center applications and services.

3. **Core Layer** interfaces the data center to external networks and connects together the various aggregation layer switches. It operates at layer-3.

Multiple redundant links connect together pairs of switches at all layers, enabling high availability at the risk of forwarding loops. Mechanisms like spanning tree construction [10] are used to break loops (like loop XYZ in Figure 2.1) in the layer-2 topology.

2.1.2 Deployment Mechanisms

In today's data centers, there is a strong coupling between the physical network topology and the *logical* topology. The logical topology refers to the sequences of special-purpose classifiers to be traversed by different types of traffic, as specified by data center policies. Current classifier deployment practices hard code these policies into the physical network

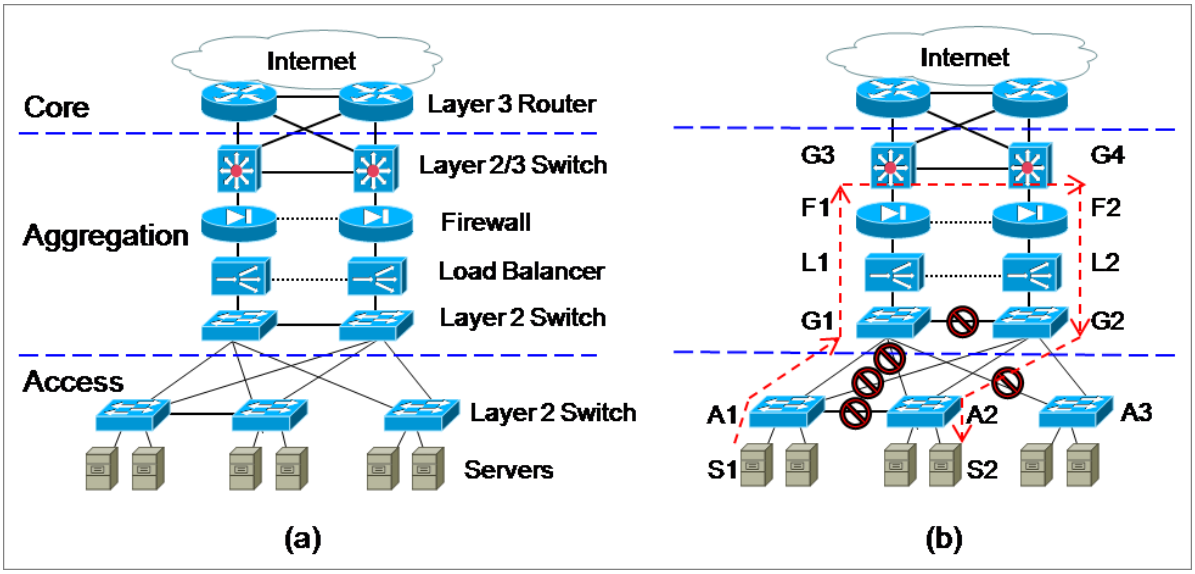

Figure 2.2. (a) On-path classifier deployment at the aggregation layer ,(b) Layer-2 path between servers *S1* and *S2* including a firewall.

topology by placing classifiers in sequence at choke points on the physical network paths. In Figure 2.2(a), a firewall and a load balancer are placed at the aggregation layer. Thus, they are on the path of all traffic entering the data center. Alternatively, classifiers can be deployed at non-chokepoints. In such cases, path selection mechanisms like spanning tree construction and other network configuration parameters like VLANs are tweaked to send traffic through the paths containing the classifiers.

The coupling between logical and physical topologies leads to classifier deployments that are hard to configure and fail to achieve three desirable properties – correctness, flexibility and efficiency. Next, we explain different classifier deployment strategies in finer detail, and illustrate how they suffer from these limitations.

2.1.3 Hard to Configure and Ensure Correctness

The process of deploying classifiers in today's data center networks is prone to misconfiguration. While literature on the practical impact and prevalence of classifier deployment issues in current data centers is scant, there is growing evidence of these problems. Accord-

ing to [7], 78% of data center downtime is caused by misconfiguration. The sheer number of misconfiguration issues cited by industry manuals [44, 10], reports of large-scale network misconfigurations [6], and anecdotal evidence from network equipment vendors and data center architects [33] complete a gloomy picture.

Configuring layer-2 switches and layer-3 routers to enforce the correct sequence of classifiers involves tweaking hundreds of knobs, a highly complex and error-prone process [7, 44, 81, 50]. Misconfiguration is exacerbated by the abundance of redundant network paths in a data center, and the unpredictability of network path selection under network churn [54, 44]. For example, the failure or addition of a network link may result in traffic being routed around the network path containing a mandatory firewall, thus violating data center security policy.

Reliance on overloading path selection mechanisms to send traffic through classifiers makes it hard to ensure that traffic traverses the correct sequence of classifiers under all network conditions. Suppose we want traffic between servers *S1* and *S2* in Figure 2.2(b) to always traverse a firewall, so that *S1* and *S2* are protected from each other when one of them gets compromised. Currently, there are three ways to achieve this:

1. Use the existing aggregation layer firewalls.
2. Deploy new firewalls.
3. Incorporate firewall functionality into the switches themselves.

All three options above are hard to implement and configure, as well as suffer from the many limitations we explain below.

Option 1: Use existing firewalls

The first option of using the existing aggregation layer firewalls requires all traffic between *S1* and *S2* to traverse the path ($S1$, $A1$, $G1$, $L1$, $F1$, $G3$, $G4$, $F2$, $L2$, $G2$, $A2$, $S2$), marked in Figure 2.2(b). An immediately obvious problem with this approach is that it

wastes resources by causing frames to gratuitously traverse two firewalls instead of one, and two load-balancers. An even more important problem is that there is no good mechanism to enforce this path between *S1* and *S2*. The following are three widely used mechanisms:

- *Remove physical connectivity:*

  By removing links $(A1, G2)$, $(A1, A2)$, $(G1, G2)$ and $(A2, G1)$, the network administrator can ensure that there is no physical layer-2 connectivity between *S1* and *S2* except via the desired path. The link $(A3, G1)$ must also be removed by the administrator or blocked out by the spanning tree protocol in order to break forwarding loops. The main drawback of this mechanism is that we lose the fault-tolerance property of the original topology, where traffic from/to *S1* can fail over to path $(G2, L2, F2, G4)$ when a classifier or a switch on the primary path (*e.g., L1*, *F1* or *G1*) fails. Identifying the subset of links to be removed from the large number of redundant links in a data center, while simultaneously satisfying different policies, fault-tolerance requirements, spanning tree convergence and classifier failover configurations, is a very complex and possibly infeasible problem.

- *Manipulate link costs:*

  Instead of physically removing links, administrators can coerce the spanning tree construction algorithm to avoid these links by assigning them high link costs. This mechanism is hindered by the difficulty in predicting the behavior of the spanning tree construction algorithm across different failure conditions in a complex highly redundant network topology [54,44]. Similar to identifying the subset of links to be removed, tweaking distributed link costs to simultaneously carve out the different layer-2 paths needed by different policy, fault-tolerance and traffic engineering requirements is hard, if not impossible.

- *Separate VLANs:*

  Placing *S1* and *S2* on separate VLANs that are inter-connected only at the aggregation-layer firewalls ensures that traffic between them always traverses a firewall. One immediate drawback of this mechanism is that it disallows applications,

clustering protocols and virtual server mobility mechanisms requiring layer-2 adjacency [3, 46]. It also forces all applications on a server to traverse the same classifier sequence, irrespective of policy. Guaranteeing classifier traversal requires all desired classifiers to be placed at all VLAN inter-connection points. Similar to the cases of removing links and manipulating link costs, overloading VLAN configuration to simultaneously satisfy many different classifier traversal policies and traffic isolation (the original purpose of VLANs) requirements is hard.

Option 2: Deploy new firewalls

The second option of using additional firewalls is also implemented through the mechanisms described above, and hence suffer the same limitations. Firewall traversal can be guaranteed by placing firewalls on every possible network path between *S1* and *S2*. However, this incurs high hardware, power, configuration and management costs. Moreover, it increases the risk of traffic traversing undesired classifiers. Apart from wasting resources, packets traversing an undesired classifier can hinder application functionality. For example, unforeseen routing changes in the Internet, external to the data center, may shift traffic to a backup ingress point with an on-path firewall that filters all non-web traffic, thus crippling other applications.

Option 3: Incorporate functionality into switches

The third option of incorporating firewall functionality into switches is in line with the industry trend of consolidating more and more classifier functionality into switches. Currently, only high-end switches [9] incorporate classifier functionality and often replace the sequence of classifiers and switches at the aggregation layer (for example, $F1$, $L1$, $G1$ and $G3$). This option suffers the same limitations as the first two, as it uses similar mechanisms to coerce *S1*-*S2* traffic through the high-end aggregation switches incorporating the required classifier functionality. These high end switches are already oversubscribed by multiple access layer switches. Sending *S1*-*S2* traffic through these switches even when a

direct path exists further strains their resources. They also become concentrated points of failure. This problem goes away if all switches in the data center incorporate all the required classifier functionality. Though not impossible, this is impractical from a cost (both hardware and management) and efficiency perspective.

2.1.4 Network Inflexibility

While data centers are typically well-planned, changes are unavoidable. For example, to ensure compliance with future regulation like Sarbanes Oxley [36], new accounting classifiers may be needed for email traffic. The dFence [76] DDOS attack mitigation classifier is dynamically deployed on the path of external network traffic during DDOS attacks. New instances of classifiers are also deployed to handle increased loads, a possibly more frequent event with the advent of on-demand instantiated virtual classifiers.

Adding a new special-purpose classifier, whether as part of a logical topology update or to reduce load on existing classifiers, currently requires significant re-engineering and configuration changes, physical rewiring of the backup traffic path(s), shifting of traffic to this path, and finally rewiring the original path. Plugging in a new classifier 'service' module into a single high-end switch is easier. However, it still involves significant re-engineering and configuration, especially if all expansion slots in the switch are filled up.

Network inflexibility also manifests as fate-sharing between classifiers and traffic flow. All traffic on a particular network path is forced to traverse the same classifier sequence, irrespective of policy requirements. Moreover, the failure of any classifier instance on the physical path breaks the traffic flow on that path. This can be disastrous for the data center if no backup paths exist, especially when availability is more important than classifier traversal.

2.1.5 Inefficient Resource Usage

Ideally, traffic should only traverse the required classifiers, and be load balanced across multiple instances of the same classifier type, if available. However, configuration inflexibility and on-path classifier placement make it difficult to achieve these goals using existing classifier deployment mechanisms. Suppose, path $(G4, F2, L2, G2)$ in Figure 2.2(b) is blocked out by spanning tree construction or explicitly blocked out by traffic engineering. All traffic entering the data center, irrespective of policy, flows through the remaining path $(G3, F1, L1, G1)$, forcing classifiers $F1$ and $L1$ to process unnecessary traffic and waste their resources. Moreover, classifiers $F2$ and $L2$ on the blocked out path remain unutilized even when $F1$ and $L1$ are struggling with overload.

2.2 In-place Classification

Many classifiers fully implement the classification operations *in-place*, *i.e.,* all by themselves. For example, a firewall implements firewall functionality all by itself. It looks up matching rules, records per-flow state, and forwards or drops the packet without consulting any other entity in the network.

Increasing classification complexity and traffic loads restrict the scalability of in-place classification. An in-place classifier is often unable to keep up with the load. Additionally, in-place classification is inefficient and hard to configure, since it does not leverage semantic context and processing resources that may be available at other network entities. In this section, we illustrate the configuration hardness and inefficiency of in-place classification using the example of a firewall load balancer.

Let us revisit the firewall load balancing scenario from Section 1.1.1, illustrated in Figure 2.3. Current firewalls are typically stateful, *i.e.,* the accept/drop decision for a packet may depend on earlier packets. For example, a firewall will accept a packet in the data TCP connection of an FTP session, only if it earlier accepted packets in the associated control connection. Packets in the reverse flow direction must also be processed by the same

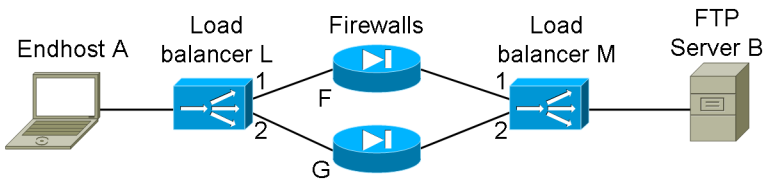

Figure 2.3. Current firewall load balancer deployment.

firewall instance. Thus, for correct firewall operations, the load balancers in Figure 2.3 must send all packets in both control and data connections of an FTP session to the same firewall instance in both flow directions.

To implement such firewall 'stickiness' for the control and data connections of an FTP session, current firewall load balancers are forced to decipher the application semantics of their traffic. This increases the complexity of load balancer design and implementation, as well as incurs additional performance overheads of gratuitous layer-7 processing.

Some load balancers rely on wiring configuration to select the same firewall instance for both traffic directions [74]. When load balancer M in Figure 2.3 receives a packet from firewall F, it stores a mapping between the packet's 5-tuple and the network interface (in this case, interface 1) on which the packet arrived. On receiving a packet in the reverse direction, it looks up the stored mapping and forwards the packet on the recorded interface. This mechanism is often ad-hoc and increases network configuration complexity.

2.3 Classification Offload

Classification *offload* is used to overcome the scalability limitations of *in-place* classification. These offload mechanisms *push* classification-related processing from classifiers to other entities with more semantic context or available processing resources per packet.

Different classification applications employ different application-specific offload solutions – for example, MPLS, Diffserv, and HTTP cookies. The lack of a generic classification

offload solution and the ad-hoc nature of existing solutions increases configuration hardness, and hampers network flexibility and efficiency.

In this section, we first describe various application-specific offload solutions. We then describe their limitations.

2.3.1 MPLS

Core routers typically have more total processing power than edge routers. However, a core router encounters substantially greater volumes of traffic per second than an edge router. As a result, a core router can afford to spend lesser time processing each packet than an edge router, inspite of having greater processing capabilities in aggregate.

Route lookup is the main classification activity of a router. Increasing number of autonomous systems and multi-homing have greatly increased routing table sizes at core routers [65, 77, 78, 80]. Consequently, the complexity of route lookup has significantly increased. Additionally, traffic engineering policies often add more complexity to a router's classification operations.

Core routers use MPLS [25] at layer 2.5 to offload expensive route lookup and traffic engineering decisions to edge routers. Using a label distribution protocol [21], network administrators configure edge routers to add different labels to packets based on their destination IP address and the traffic engineering policies they match. The label distribution protocol also establishes next hop tables mapping labels to output interfaces at core routers. A downstream core router can thus easily determine a packet's next hop by looking up its label in its next hop table. Exact matching based label lookup is much simpler than longest prefix matching based route and traffic policy lookup. In this way, a core router improves its scalability by shedding some of its classification load to edge routers.

2.3.2 Diffserv

Most routers can provide different qualities of service (QoS) to different types of traffic. For example, a router may accord high forwarding priority to VoIP traffic and low priority to data backup traffic. To provide different QoS to a packet, a router must first classify its traffic type and retrieve the appropriate QoS policy. Routers commonly identify traffic types by looking at packet transport port numbers or by looking deep into packet payloads. These are complex operations that hinder the router's scalability.

Routers use Diffserv [2] to push the identification of a packet's QoS requirements to endhosts and ingress routers. Endhosts and ingress routers use local application context, QoS policies and traffic statistics to determine a packet's QoS requirements. They convey the packet's QoS class to downstream routers by setting the DSCP bits in its layer-3 (IP) header. A downstream router simply uses the DSCP bits to determine how the packet should be forwarded. As in the case of MPLS, it improves scalability by avoiding complex rule matching.

2.3.3 HTTP Cookies

An HTTP session may consist of multiple HTTP and HTTPS connections. For example, an HTTP session involving an online shopping experience includes multiple HTTP connections for browsing and adding items to the shopping cart, and HTTPS connections for payment and shipping information. If the application servers are stateful, then a load balancer must send all connections in a session to the same server instance for semantic correctness. Otherwise, shopping cart contents may be corrupted, or the server may lack the correct SSL connection parameters. Even if application servers are stateless, sending all connections to the same instance facilitates performance improvements through caching.

One simple strategy for the load balancer is to send all packets with the same source IP address to the same application server instance. This strategy has two limitations:

1. Different connections within an HTTP session may not have the same source IP

address. TCP connections from hosts in a home ISP network are often load balanced across multiple proxies or NATs. A proxy or NAT rewrites a packet's source IP address with its globally routable IP address. Hence, the simple of strategy of relying on source IP addresses does not ensure that the same server instance is selected for all connections in a session. This problem is often called the *AOL Mega Proxy problem* [40, 11].

2. Relying solely on source IP addresses may result in coarse-grained load balancing and cause load skew. Connections from a large number of hosts behind a NAT or proxy will have the same source IP address. Sending all these connections to the same server may overload it.

Since source IP addresses are insufficient, load balancers currently rely on direct endhost support. A load balancer uses HTTP cookies [15] to offload HTTP session identification to the endhosts. When the load balancer sees an HTTP request without a cookie, it inserts a new cookie into the HTTP response. The cookie identifies the webserver instance selected for this session. The application, say web browser, at the endhost has local and semantic context to identify the multiple HTTP/HTTPS connections in an HTTP session. It includes the same cookie value in subsequent connections of the session. The load balancer reads the cookie value by reconstructing the transport stream and parsing the application, *i.e.,* layer-7 headers. From the cookie value, it determines the server instance to be used for the connection.

2.3.4 Limitations

Each of the classification offload mechanisms described above is a point solution tailored to a particular application and operating at a single layer. The following are their three main limitations:

1. **Configuration Hardness**

   A network administrator must separately implement and configure the different clas-

sification offload mechanisms corresponding to different applications. This increases network configuration complexity, and increases the probability for errors arising from the interactions between different mechanisms.

2. **Inefficiency**

   Some offload solutions rely on expensive deep packet inspection. For instance, a load balancer must reconstruct the transport stream and parse layer-7 HTTP headers to simply read the HTTP cookie value. Such expensive operations are not a fundamental component of classification offload, but are simply inefficiencies of the current mechanisms.

3. **Inflexibility**

   Current offload mechanisms are inflexible since they are tailored to a specific application and protocol layer. A new classification application will require re-inventing and deploying similar offload mechanisms. Due to the difficulty and costs of such re-design efforts, many applications are deprived of the benefits of classification offload.

2.4 Limited Understanding of Classifier Operations

Current packet classifiers are complex devices that implement a wide range of functionality at different layers - from simple layer-3 router lookup to complex layer-7 HTTP session identification. Different classifiers interact with the network in different ways. They have different expectations about the type of packets reaching it. For example, a load balancer often requires packets to be explicitly IP addressed to it. A classifier may change the packet header and contents as part of its operation. For example, load balancers often rewrite the destination IP address and port number of a packet to those of the selected webserver. Moreover, a classifier often has multiple operation modes, each with its own configuration and requirements. For example, a load balancer can be deployed in two-legged or single-legged mode as shown in Figure 2.4. In the two-legged mode, it often rewrites the source IP address and port number to its own IP address and a local port number. On the other

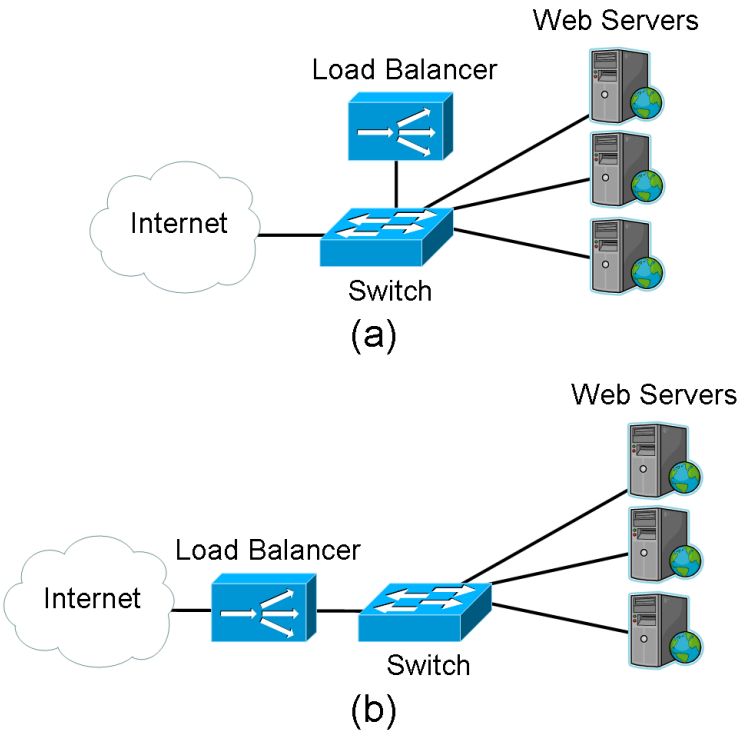

Figure 2.4. Different load balancer deployment configurations: (a) Single-legged, (b) Two-legged

hand, in the single-legged Direct Server Return (DSR) [74] mode, packets from the web server directly flow to the clients, bypassing the load balancer. Thus the load balancer does not rewrite the source IP address and port number in this mode.

Information about such classifier requirements and behavior is not readily available. Such information is often buried inside big configuration manuals. There is currently no standard way to succinctly describe classifiers. In many cases, the information is simply not available.

The non-availability of this information makes network planning, deployment and configuration hard. For example, a network administrator cannot easily determine how two classifiers placed in sequence will interact – Do the packets forwarded by the upstream classifier satisfy the packet header expectations of the downstream classifier? Without clear information about classifier behavior, it is difficult to correctly configure the network, as well as to detect anomalous behavior and troubleshoot the network.

The lack of a concise and standard language to describe different classifiers also hinders classifier-related research. In our own research experience designing and implementing the *policy-aware switching layer* (Section 5), the non-availability of clear information about how some classifiers processed packets led to some initial wrong design decisions which later manifested as hard-to-debug errors. We describe our experience in detail in Section 3.2.3.

2.5 Summary

In this chapter, we described the current classifier deployment and classification offload mechanisms. These mechanisms, as well as in-place classification, suffer from three common limitations – configuration hardness, inefficiency and inflexibility. The lack of clear knowledge about classifier functionality and configuration requirements hinders network deployments as well as networking research.

In the next chapter, we present a classifier model that helps us better understand classifiers. The model sets the stage for our solutions that tackle the limitations, in subsequent chapters.

Chapter 3

Modeling Classifiers

There is great diversity in how today's classifiers process and transform packets, and in how they are configured and deployed. For example, a firewall is commonly connected inline on the physical network path, and transparently forwards packets unmodified or drops them. A load balancer, on the other hand, rewrites packet headers and contents, and often requires packets to be explicitly IP addressed and forwarded to it.

Correctly deploying and configuring classifiers is a challenging task by itself. Without a clear understanding of how different classifiers process packets and interact with the network and with other classifiers, network planning, verification of operational correctness and troubleshooting become even more complicated. As we described in the previous chapter, the lack of a concise standard language to describe classifier operations and configurations hinders existing network deployments as well as classifier-related research.

In this chapter, we present a general *model* to describe classifier functionality and deployment configurations. We envision an online repository containing models of commonly used classifiers. Network administrators can use models downloaded from this repository to plan network topologies and to validate if the observed network traffic matches the patterns described by the model. The standard and concise language offered by our model helps researchers quickly understand diverse classifier functionality, and how it affects their

research. To ease model construction, we have prototyped a tool that infers hints about a particular classifier's operations through blackbox testing. We have also prototyped a tool that validates a classifier's operations against its model, and thus helps detect unexpected behavior.

Next, we describe our model and illustrate its details using three common classifiers – firewall, layer-4 load balancer, and SSL-enabled layer-7 load balancer; and using different kinds of Network Address Translator (NAT) [39] boxes. We then demonstrate the utility of our model using multiple examples in Section 3.2. Section 3.3 describes the limitations of our model, while section 3.4 compares our model to related work.

3.1 The Model

A classifier in our model consists of *zones*, *input pre-conditions*, *state databases*, *processing rules*, *auxiliary traffic*, and the *interest* and *state fields* deduced from the *processing rules*. In this section, we describe and illustrate our model using three common classifiers – firewall, layer-4 load balancer and SSL-offload capable layer-7 load balancer, and using different kinds of NATs. Although NATs do not themselves classify packets, they implement packet transformations similar to classifiers, as well have similar deployment requirements.

Table 3.1 describes the notations used in our model.

3.1.1 Interfaces and Zones

A classifier has one or more physical network interfaces. Packets enter and exit a classifier through one or more of these *interfaces*. Each physical interface is associated with one or more logical network *zones*. A *zone* represents a packet entry and exit point from the perspective of classifier functionality. A classifier processes packets differently based on their ingress and egress zones.

For example, the firewall shown in Figure 3.1(a) has two physical interfaces, one belonging to the `red` zone that represents the insecure external network, and the other belonging

Table 3.1. Notations used in our model

$\wedge$	logical AND operation
!	logical NOT operation
`sm`	source MAC (layer-2) address
`dm`	destination MAC (layer-2) address
`si`	source IP (layer-3) address
`di`	destination IP (layer-3) address
`sp`	source TCP/UDP (layer-4) port
`dp`	destination TCP/UDP (layer-4) port
p	packet
$[hd]$	packet with header h and payload d
5tpl	packet 5-tuple, *i.e.,* `si`,`di`, `sp`, `dp`, `proto`
X^{rev}	Swaps any source-destination IP, MAC or port number pairs in X
$\mathcal{Z}(A, p)$	`true` if packet p arrived at or departed zone A
$\mathcal{I}(\mathcal{P}, p)$	Input pre-condition; `true` if packet p matches pattern $\mathcal{P}$
$\mathcal{C}(p)$	condition specific to classifier functionality
newflow?(p)	`true` if packet p indicates a new flow, *e.g.,* TCP SYN
`set`$(A, \text{key} \rightarrow \text{val})$	Stores the specified key-value pair in zone A's state database
$\mathcal{S}$: `get`?(A, key)	Returns `true` and assigns val to $\mathcal{S}$ if key $\rightarrow$ val is present in zone A's state database

to the `green` zone representing the secure internal network. Packets entering through the `red` zone are more stringently checked than those entering through the `green` zone.

Similarly, the NAT in Figure 3.1(b) has two different physical network interfaces, one belonging to the internal network (zone `int`) and the other belonging to the external network (zone `ext`). The source IP and port number are rewritten for packets received at zone `int`, while the destination IP and port number are rewritten for packets received at zone `ext`. Figure 3.1(c) shows a load balancer with a single physical network interface that belongs to two different zones. Zone `inet` represents the Internet and zone `srvr` represents the web server farm. The load balancer forwards packets received at zone `inet` to webserver instances in zone `srvr`.

The mapping between interfaces and zones is pre-determined by the classifier vendor or configured during classifier initialization. The zone of a frame reaching an interface belonging to multiple zones is identified by its VLAN tags, IP addresses and/or transport port numbers.

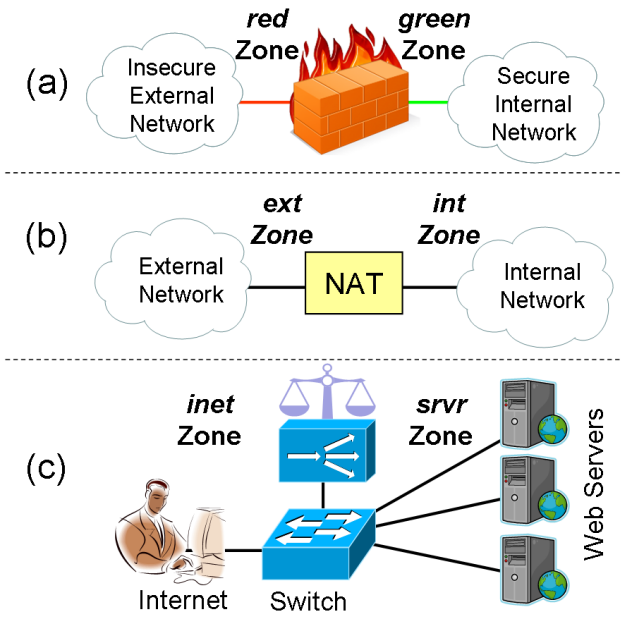

Figure 3.1. Zones of different classifiers: (a) firewall (b) NAT (c) load balancer in single-legged configuration

3.1.2 Input Pre-conditions

A classifier may only process certain types of packets. *Input pre-conditions* record the types of packets which are accepted by a classifier for processing. For example, a transparent firewall processes all packets received by it, whereas a load balancer in a single-legged configuration processes a packet arriving at its `inet` zone only if the packet is explicitly addressed to it at layers 2, 3 and 4. Similarly, a NAT processes all packets received at its `int` zone, but requires those received at its `ext` zone to be addressed to it at layers 2 and 3.

We represent input pre-conditions using a clause of the form $\mathcal{I}(\mathcal{P}, p)$, which is `true` if the headers and contents of packet p match the pattern $\mathcal{P}$. For example, the firewall has the input pre-condition $\mathcal{I}(< * >, p)$ and the load balancer has $\mathcal{I}(<$ `dm` $= \mathtt{MAC}_{\mathrm{LB}},$ `di` $= \mathtt{IP}_{\mathrm{LB}},$ `dp` $= 80 >, p)$ for its `inet` zone, where $\mathtt{MAC}_{\mathrm{LB}}$ and $\mathtt{IP}_{\mathrm{LB}}$ are the load balancer's layer-2 and layer-3 addresses. Although $\mathcal{I}(< * >, p)$ is a tautology, we explicitly specify it in order to enhance model clarity.

3.1.3 State Database

Many classifiers maintain state associated with the flows and sessions they process. We represent state in our model using key-value pairs stored in zone-independent or zone-specific *state databases*. Processing rules (described next) use the `set` and `get`? primitives to manipulate state.

Accurately tracking state removal is hard, unless explicitly specified by the `del` primitive in a processing rule. We use special processing rules to flag discrepancies that may result from state expiration, instead of using state expiration timeouts. State expiration timeouts are ineffective due to inaccuracies in timeout values or in their fine-grained measurement.

3.1.4 Processing Rules

We model the functionality of a classifier using *Processing Rules*. A processing rule specifies what a classifier does on receiving a packet or when a condition becomes true. For example, the processing of an incoming packet is represented by a rule of the general form:

$$\boxed{\mathcal{Z}(A,p) \wedge \mathcal{I}(\mathcal{P},p) \wedge \mathcal{C}(p) \quad \Longrightarrow \quad \mathcal{Z}(B,\mathcal{T}(p)) \wedge \text{state ops}}$$

The above rule indicates that a packet p reaching zone A of the classifier is transformed to $\mathcal{T}(p)$ and emitted out through zone B. The packet is emitted out only if it satisfies the input pre-condition $\mathcal{I}(\mathcal{P},p)$ and a classifier-specific condition $\mathcal{C}(p)$. The rule also indicates that the classifier may manipulate state associated with the TCP flow or application session to which the packet belongs. We now present concrete examples of processing rules for common classifiers.

Firewall

We start with a simple stateless layer-4 firewall. The firewall either drops a packet received on its `red` zone or relays it unmodified to the `green` zone. Our model captures this behavior using the following two rules:

$$\boxed{\begin{array}{ccc} \mathcal{Z}(\texttt{red}, p) \wedge \mathcal{I}(<*>, p) \wedge \mathcal{C}_{\text{accept}}(p) & \Longrightarrow & \mathcal{Z}(\texttt{green}, p) \\ \mathcal{Z}(\texttt{red}, p) \wedge \mathcal{I}(<*>, p) \wedge \mathcal{C}_{\text{drop}}(p) & \Longrightarrow & \text{DROP}(p) \end{array}}$$

Since $\mathcal{I}(<*>, p)$ is a tautology, whether a packet is dropped or accepted by the firewall is solely determined by the $\mathcal{C}_{\text{accept}}$ and $\mathcal{C}_{\text{drop}}$. These clauses represent the firewall's basic filtering functionality. Common filtering rules can be easily represented using simple boolean expressions – *e.g.,* $\mathcal{C}_{\text{accept}}(p) : p.\texttt{di} = 80 \;||\; p.\texttt{si} = 128.34.45.6$. We leverage external models like the Unified Firewall Model [79] to construct C clauses for more complex rules. Rules for packets in the `green` $\rightarrow$ `red` direction are similar.

NAT

Next, we show how different types of NATs can be represented in our model. Although NATs are not classifiers themselves, they are similar to classifiers in terms of packet transformations and deployment configurations. Unlike the firewall in the previous example, a NAT rewrites packet headers and maintains per-flow state. We first describe the processing rules for a *full cone* NAT below. We then change it to represent a *symmetric* NAT.

Full Cone NAT			
(i)	$\mathcal{Z}(\texttt{int}, [hd])$ $\wedge\mathcal{I}(< * >, [hd])$ $\wedge!\mathcal{S}$: $\texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}])$	$\Longrightarrow$	$\mathcal{Z}(\texttt{ext}, [\text{SNAT}_{\text{fwd}}(h, \texttt{newport})d])$ $\wedge\texttt{set}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}] \rightarrow \texttt{newport})$ $\wedge\texttt{set}(\texttt{ext}, \texttt{newport} \rightarrow [h.\texttt{si}, h.\texttt{sp}])$
	$\text{SNAT}_{\text{fwd}}([\texttt{sm}, \texttt{dm},$ $\texttt{si}, \texttt{di}, \texttt{sp}, \texttt{dp}], \texttt{PORT})$	$=$	$[\texttt{MAC}_{\text{NAT}}, \texttt{MAC}_{\text{gw}}, \texttt{IP}_{\text{NAT}}, \texttt{di}, \texttt{PORT}, \texttt{dp}]$
(ii)	$\mathcal{Z}(\texttt{int}, [hd])$ $\wedge\mathcal{I}(< * >, [hd])$ $\wedge\mathcal{S}$: $\texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}])$	$\Longrightarrow$	$\mathcal{Z}(\texttt{ext}, [\text{SNAT}_{\text{fwd}}(h, \mathcal{S})d])$
(iii)	$\mathcal{Z}(\texttt{ext}, [hd])$ $\wedge\mathcal{I}(< \texttt{di} = \texttt{IP}_{\text{NAT}},$ $\texttt{dm} = \texttt{MAC}_{\text{NAT}} >, [hd])$ $\wedge\mathcal{S} : \texttt{get?}(\texttt{ext}, h.\texttt{dp})$	$\Longrightarrow$	$\mathcal{Z}(\texttt{int}, [\text{SNAT}_{\text{rev}}(h, \mathcal{S}.\texttt{si}, \mathcal{S}.\texttt{sp})d])$
	$\text{SNAT}_{\text{rev}}([\texttt{sm}, \texttt{dm}, \texttt{si}, \texttt{di},$ $\texttt{sp}, \texttt{dp}], \texttt{IP}, \texttt{PORT})$	$=$	$[\texttt{MAC}_{\text{NAT}}, \texttt{MAC}_{\texttt{IP}}, \texttt{si}, \texttt{IP}, \texttt{sp}, \texttt{PORT}]$
(iv)	$\mathcal{Z}(\texttt{int}, [hd])$ $\wedge\mathcal{I}(< * >, [hd])$ $\wedge\mathcal{S}$: $\texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}])$	$\Longrightarrow$	$\text{DROP}([hd])$ $\wedge\text{WARN}(\text{inconsistent state})$
(v)	$\mathcal{Z}(\texttt{ext}, [hd])$ $\wedge\mathcal{I}(< \texttt{di} = \texttt{IP}_{\text{NAT}},$ $\texttt{dm} = \texttt{MAC}_{\text{NAT}} >, [hd])$ $\wedge\mathcal{S} : \texttt{get?}(\texttt{ext}, h.\texttt{dp})$	$\Longrightarrow$	$\text{DROP}([hd])$ $\wedge\text{WARN}(\text{inconsistent state})$

- **Rule (i)** describes how a full cone NAT processes a packet $[hd]$ with a previously unseen $[\texttt{si}, \texttt{sp}]$ pair received at its `int` zone. It allocates a new port number using a standard mechanism like random or sequential selection, or using a custom mechanism beyond the scope of our model. It stores $[\texttt{si}, \texttt{sp}] \rightarrow \texttt{newport}$ and $\texttt{newport} \rightarrow [\texttt{si}, \texttt{sp}]$ in zone `int`'s and zone `ext`'s state databases, respectively. It rewrites the packet header h by applying the source NAT (SNAT_{fwd}) transformation function – the source MAC and IP addresses are replaced with the NAT's own publicly visible addresses, the source port with the newly allocated port number, and the destination MAC with the NAT's next hop IP gateway. It then emits the packet with the rewritten header and unmodified payload through its `ext` zone.

- **Rule (ii)** specifies that the NAT emits a packet with a previously seen $[\texttt{si}, \texttt{sp}]$ pair through zone `ext`, after applying SNAT_{fwd} with the port number recorded in rule (i).

- **Rule (iii)** describes how the NAT processes a packet reaching the `ext` zone. It first retrieves the `newport` $\rightarrow$ $[\texttt{si}, \texttt{sp}]$ state recorded in rule (i) using the packet's destination port number. It then applies the reverse source NAT transformation function(SNAT_{rev}) and emits the modified packet through zone `int`.

- **Rule (iv)** and **Rule (v)** flag discrepancies resulting from the model's inaccuracy in tracking state expiration. The NAT may drop a packet arriving at its `int` or `ext` zone if the state associated with the packet has expired without the model's knowledge.

Unlike a full cone NAT, a symmetric NAT allocates a separate port for each $[\texttt{si}, \texttt{sp}, \texttt{di}, \texttt{dp}]$ tuple seen at its `int` zone, rather than for each $[\texttt{si}, \texttt{sp}]$ pair. Thus, for a symmetric NAT, the zone `int` state set in rule (i) and retrieved in rules (ii) and (iv), is keyed by $[h.\texttt{si}, h.\texttt{sp}, h.\texttt{di}, h.\texttt{dp}]$ rather than by just $[h.\texttt{si}, h.\texttt{sp}]$. A symmetric NAT is also more restrictive than a full cone NAT. It relays a packet with header $[\texttt{IP}_s, \texttt{IP}_{\text{NAT}}, \texttt{PORT}_s, \texttt{PORT}_d]$ from the `ext` zone only if it had earlier received a packet destined to $\texttt{IP}_s : \texttt{PORT}_s$ at the `int` zone and had rewritten its source port to $\texttt{PORT}_d$. This restrictive behavior is captured by keying the zone `ext` state set in rule (i) and retrieved in rules (iii) and (v) with $[h.\texttt{di}, h.\texttt{dp}, \texttt{newport}]$ rather than with just `newport`. Other NAT types like *restricted cone* and *port restricted cone* can be easily represented with similar minor modifications.

Restricted Cone NAT

$$\begin{array}{l|l}
\mathcal{Z}(\texttt{int}, [hd]) & \\
\wedge\mathcal{I}(< * >, [hd]) & \mathcal{Z}(\texttt{ext}, [\text{SNAT}_{\text{fwd}}(h, \texttt{newport})d]) \\
\wedge !\texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}]) & \wedge\texttt{set}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}] \rightarrow \texttt{newport}) \\
 & \wedge\texttt{set}(\texttt{ext}, [h.\texttt{di}, \texttt{newport}] \rightarrow [h.\texttt{si}, h.\texttt{sp}]) \\
 & \\
\mathcal{Z}(\texttt{int}, [hd]) & \\
\wedge\mathcal{I}(< * >, [hd]) & \mathcal{Z}(\texttt{ext}, [\text{SNAT}_{\text{fwd}}(h, \mathcal{S}.\texttt{PORT})d]) \\
\wedge\mathcal{S} : \texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}]) & \wedge\texttt{set}(\texttt{ext}, [h.\texttt{di}, \mathcal{S}.\texttt{PORT}] \rightarrow [h.\texttt{si}, h.\texttt{sp}]) \\
 & \\
\mathcal{Z}(\texttt{ext}, [hd]) & \\
\wedge\mathcal{I}(< \texttt{di} = \texttt{IP}_N, & \\
\texttt{dm} = \texttt{MAC}_N >, [hd]) & \mathcal{Z}(\texttt{int}, [\text{SNAT}_{\text{rev}}(h, \mathcal{S}.\texttt{si}, \mathcal{S}.\texttt{sp})d]) \\
\wedge\mathcal{S} : \texttt{get?}(\texttt{ext}, [h.\texttt{si}, h.\texttt{dp}]) &
\end{array}$$

Port Restricted Cone NAT	
$\mathcal{Z}(\texttt{int}, [hd])$ $\wedge\mathcal{I}(< * >, [hd])$ $\wedge!\texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}])$	$\Longrightarrow$ $\mathcal{Z}(\texttt{ext}, [\text{SNAT}_{\text{fwd}}(h, \texttt{newport})d])$ $\wedge\texttt{set}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}] \rightarrow \texttt{newport})$ $\wedge\texttt{set}(\texttt{ext}, [h.\texttt{di}, h.\texttt{dp}, \texttt{newport}]$ $\rightarrow [h.\texttt{si}, h.\texttt{sp}])$
$\mathcal{Z}(\texttt{int}, [hd])$ $\wedge\mathcal{I}(< * >, [hd])$ $\wedge\mathcal{S} : \texttt{get?}(\texttt{int}, [h.\texttt{si}, h.\texttt{sp}])$	$\Longrightarrow$ $\mathcal{Z}(\texttt{ext}, [\text{SNAT}_{\text{fwd}}(h, \mathcal{S}.\texttt{PORT})d])$ $\wedge\texttt{set}(\texttt{ext}, [h.\texttt{di}, h.\texttt{dp}, \mathcal{S}.\texttt{PORT}]$ $\rightarrow [h.\texttt{si}, h.\texttt{sp}])$
$\mathcal{Z}(\texttt{ext}, [hd])$ $\wedge\mathcal{I}(< \texttt{di} = \texttt{IP}_N,$ $\texttt{dm} = \texttt{MAC}_N >, [hd])$ $\wedge\mathcal{S} : \texttt{get?}(\texttt{ext}, [h.\texttt{si}, h.\texttt{sp}, h.\texttt{dp}])$	$\Longrightarrow \mathcal{Z}(\texttt{int}, [\text{SNAT}_{\text{rev}}(h, \mathcal{S}.\texttt{si}, \mathcal{S}.\texttt{sp})d])$

Layer-4 load balancer

Here, we present a layer-4 load balancer. The load balancer rewrites a packet's destination IP address to that of an available server.

Layer-4 Load Balancer		
(i)	$\mathcal{Z}(\texttt{inet}, [hd])\wedge$ $\mathcal{I}(< \texttt{dm} = \texttt{MAC}_{\text{LB}}, \texttt{di} = \texttt{IP}_{\text{LB}},$ $\texttt{dp} = 80 >, [hd])$ $\wedge\text{newflow?}([hd])$	$\Longrightarrow$ $\mathcal{Z}(\texttt{srvr}, [\text{DNAT}_{\text{fwd}}(h, W_i)d])$ $\wedge\texttt{set}(\texttt{inet}, h.5\text{tpl} \rightarrow W_i)$ $\wedge\texttt{set}(\texttt{srvr}, \text{DNAT}_{\text{fwd}}(h, W_i)^{\text{rev}}.5\text{tpl} \rightarrow \texttt{true})$
	$\text{DNAT}_{\text{fwd}}([\texttt{sm}, \texttt{dm}, \texttt{si}, \texttt{di},$ $\texttt{sp}, \texttt{dp}], W)$	$= [\texttt{sm}, \texttt{MAC}_W, \texttt{si}, \texttt{IP}_W, \texttt{sp}, \texttt{dp}]$
(ii)	$\mathcal{Z}(\texttt{inet}, [hd])\wedge$ $\mathcal{I}(< \texttt{dm} = \texttt{MAC}_{\text{LB}}, \texttt{di} = \texttt{IP}_{\text{LB}},$ $\texttt{dp} = 80 >, [hd])$ $\wedge!\text{newflow?}([hd])$ $\wedge\mathcal{S} : \texttt{get?}(\texttt{inet}, h.5\text{tpl})$	$\Longrightarrow \mathcal{Z}(\texttt{srvr}, [\text{DNAT}_{\text{fwd}}(h, \mathcal{S})d])$
(iii)	$\mathcal{Z}(\texttt{srvr}, [hd])\wedge$ $\mathcal{I}(< \texttt{sm} = \texttt{MAC}_{W_i}, \texttt{si} = \texttt{IP}_{W_i},$ $\texttt{sp} = 80 >, [hd])$ $\wedge\mathcal{S} : \texttt{get?}(\texttt{srvr}, h.5\text{tpl})$	$\Longrightarrow \mathcal{Z}(\texttt{inet}, [\text{DNAT}_{\text{rev}}(h)d])$
	$\text{DNAT}_{\text{rev}}([\texttt{sm}, \texttt{dm}, \texttt{si}, \texttt{di}, \texttt{sp}, \texttt{dp}])$	$= [\texttt{MAC}_{\text{LB}}, \texttt{MAC}_{\text{gw}}, \texttt{IP}_{\text{LB}}, \texttt{di}, \texttt{sp}, \texttt{dp}]$

- **Rule (i)** describes how the load balancer processes the first packet of a new flow received at its `inet` zone. The load balancer selects a webserver instance W_i for the flow and records it in the `inet` zone's state database. It rewrites the destination

IP and MAC addresses of the packet to W_i using the destination NAT (DNAT_{fwd}) transformation function. It then emits the packet out through the `srvr` zone. It also records this flow in the `srvr` zone's state database, keyed by the 5-tuple of the packet expected in the reverse flow direction.

- **Rule (ii)** specifies that subsequent packets of the flow will simply be emitted out after rewriting the destination IP and MAC addresses to those of the recorded webserver instance.

- **Rule (iii)** describes the processing of a packet received from a webserver. The load balancer verifies the existence of flow state for the packet and then emits it out through the `inet` zone after applying the reverse DNAT transformation – *i.e.,* rewriting the source IP and MAC addresses to those of the load balancer and the destination MAC to the next hop IP gateway.

The webserver instance selection mechanism is beyond the scope of our general model. However, the load balancer model can be augmented with primitives to represent common selection mechanisms like *least loaded* and *round-robin*. In the example above, we assumed that the load balancer was set as the default IP gateway at each webserver. Other deployment configurations (*e.g.,* direct server return or source NAT) can be represented with minor modifications.

Layer-7 load balancer

We illustrate how our model describes a classifier whose processing spans both packet headers and contents, and is not restricted to one-to-one packet transformations. We use a layer-7 SSL-offload capable load balancer as an example. The layer-7 load balancer acts as the end point of the TCP connection from a client (the CL connection). Since modeling TCP behavior is very hard, we abstract it using a blackbox TCP state machine $\texttt{tcp}_{CL}$. We buffer the data received from the client in a byte queue $\texttt{D}_{CL}$. The $\mathcal{I}$ clauses are similar to those of the layer-4 load balancer and hence not repeated below:

Layer-7 Load Balancer	
(i) $\mathcal{Z}(\texttt{inet}, [hd])$ $\wedge\mathcal{I}(\cdots)$ $\wedge$newflow?$([hd])$	$\Longrightarrow \texttt{set}(\texttt{inet}, h.5\text{tpl} \rightarrow [\texttt{tcp}_{CL} = \text{TCP.new}, \texttt{D}_{CL} = \text{Data.new}, \texttt{h}_{CL} = h])$
(ii) $\mathcal{Z}(\texttt{inet}, [hd])$ $\wedge\mathcal{I}(\cdots)$ $\wedge$!newflow?$([hd])$ $\wedge\mathcal{S}$: get?$(\texttt{inet}, h.5\text{tpl})$	$\Longrightarrow \mathcal{S}.\texttt{tcp}_{CL}.\text{recv}(h)$ $\wedge\mathcal{S}.\texttt{D}_{CL} + d$
(iii) $\mathcal{S}.\texttt{tcp}_{CL}$.ready?	$\Longrightarrow \mathcal{Z}(\texttt{inet}, \mathcal{S}.\texttt{tcp}_{CL}.\text{send}(\mathcal{S}.\texttt{h}_{CL}^{\text{rev}}, \mathcal{S}.\texttt{D}_{LS}.\text{read}))$
(iv) $\mathcal{S}.\texttt{D}_{CL}$.url?	$\Longrightarrow \mathcal{S}.\texttt{h}_{LS} = \text{DNAT}_{\text{fwd}}(\mathcal{S}.\texttt{h}_{CL}, W_i)$ $\wedge\texttt{set}(\texttt{srvr}, \mathcal{S}.\texttt{h}_{LS}^{\text{rev}}.5\text{tpl} \rightarrow \mathcal{S})$ $\wedge\mathcal{S}.\texttt{D}_{LS} = \text{Data.new}$ $\wedge\mathcal{S}.\texttt{tcp}_{LS} = \text{TCP.new}$
(v) $\mathcal{Z}(\texttt{srvr}, [hd])$ $\wedge\mathcal{I}(\cdots)$ $\wedge\mathcal{S}$: get?$(\texttt{srvr}, h.5\text{tpl})$	$\Longrightarrow \mathcal{S}.\texttt{tcp}_{LS}.\text{recv}(h)$ $\wedge\mathcal{S}.\texttt{D}_{LS} + d$
(vi) $\mathcal{S}.\texttt{tcp}_{LS}$.ready?	$\Longrightarrow \mathcal{Z}(\texttt{srvr}, \mathcal{S}.\texttt{tcp}_{LS}.\text{send}(\mathcal{S}.\texttt{h}_{LS}, \mathcal{S}.\texttt{D}_{CL}.\text{read}))$

- **Rule (i)** specifies that the load balancer creates $\texttt{tcp}_{CL}$ and $\texttt{D}_{CL}$ and records them along with the packet header, on receiving the first packet of a new flow from a client at the `inet` zone.

- **Rule (ii)** specifies how the TCP state and data queue of the CL connection are updated as packets of an existing flow arrive from the client.

- **Rule (iii)**, triggered when $\texttt{tcp}_{CL}$ has data or acks to send, specifies that packets from the load balancer to the client will have header $\texttt{h}_{CL}^{\text{rev}}$ (with appropriate sequence numbers filled in by $\texttt{tcp}_{CL}$) and payload read from the $\texttt{D}_{LS}$ queue, if it was already created by the firing of rule (iv).

- **Rule (iv)**, triggered when the data collected in $\texttt{D}_{CL}$ is sufficient to parse the HTTP request URL and/or cookies, specifies that the load balancer selects a webserver instance W_i and opens a TCP connection to it, *i.e.,* creates $\texttt{tcp}_{LS}$ and $\texttt{D}_{LS}$. It also

installs a pointer to the state indexed by the DNATed header $\mathtt{h}_{LS}$ in the `srvr` zone's state database.

- **Rule (v)** shows how this state is retrieved, and its $\mathtt{tcp}_{LS}$ and $\mathtt{D}_{LS}$ updated, on receipt of a packet from a webserver.

- **Rule (vi)** specifies the header and payload of packets sent by the load balancer to a webserver instance – $\mathtt{h}_{LS}$ and data read from $\mathtt{D}_{CL}$.

The rules listed above represent a plain layer-7 load balancer. We can represent an SSL-offload capable load balancer by replacing the + and `read` data queue operations with $+_{\text{ssl}}$ and $\mathtt{read}_{\text{ssl}}$ operations that perform SSL encryption and decryption on the data. Such a change does not disturb other rules. Similar to the TCP blackbox, we abstract out SSL protocol details.

3.1.5 Auxiliary Traffic

A classifier may generate additional traffic, apart from its core packet transformation and forwarding functionality. For example, a load balancer periodically checks the liveness of its target servers by making TCP connections to each server. It may also send out an ARP requests on receiving packet. Such packets which are not part of the classifier's core functionality, but support it, are referred to as *auxiliary traffic* in our model.

Our model represents auxiliary traffic using processing rules. For example, the auxiliary traffic associated with the load balancer is:

Load Balancer Auxiliary Traffic	
PERIODIC	$\Longrightarrow \mathcal{Z}(\mathtt{srvr}, \text{PROBE}(\mathtt{IP}_{W_i}))$
$\mathcal{Z}(\mathtt{inet}, [hd])$ $\wedge \mathcal{S} : \mathtt{get?}(\mathtt{inet}, h.5\text{tpl})$ $\wedge !\mathcal{S}' : \mathtt{get?}(-, \mathtt{IP}_{\mathcal{S}})$	$\Longrightarrow \mathcal{Z}(\mathtt{srvr}, \text{ARPREQ}(\mathtt{IP}_{\mathcal{S}}))$
$\mathcal{Z}(\mathtt{srvr}, \text{ARPRPLY}(\mathtt{IP}, \mathtt{MAC}))$	$\Longrightarrow \mathtt{set}(-, \mathtt{IP} \rightarrow \mathtt{MAC})$

The PROBE function returns a set of packets to check the liveness of server W_i. In the

simple case, these are just TCP hand-shake packets with the appropriate `sm`, `dm`, `si`, `di`, `sp` and `dp`.

3.1.6 Interest and State fields

The *interest fields* of a classifier denote the packet fields it reads or modifies. The *state fields* include the interest fields used by the classifier in storing and retrieving state. These fields can be easily deduced from the processing rules. However, they are explicitly presented in the model as they can highlight unexpected classifier behavior (Section 3.2.3).

3.2 Utility of a Classifier Model

In this section, we first explain how we constructed models for many real-world classifiers. We then describe how our model helps in planning and troubleshooting existing classifier deployments, and in guiding the development of new network architectures.

3.2.1 Model Instances

The models for the firewall, NAT, and layer-4 and layer-7 load balancers illustrated in the previous section were constructed by analyzing generic classifier descriptions and taxonomies(like RFC 3234 [24]), by referring vendor manuals, and by observing the following real-world classifiers and NATs in deployment:

- Linux Netfilter/iptables software firewall [41]
- Netgear home NAT [28]
- BalanceNg layer-4 software load balancer [4]
- HAProxy layer-7 load balancer Vmware appliance [14]

We have prototyped a blackbox testing based model inference tool to aid model construction. The tool infers hints about a classifier's operations by carefully sending different

kinds of packets on one zone and observing the packets emerging from other zones, as illustrated in Figure 3.2. The following are some of the inferences generated by it:

1. The layer-4 load balancer remembers source MAC addresses of packets processed by it in the `inet` → `srvr` direction and uses them in packets in the reverse direction. The tool made this inference by correlating rewritten packet header fields with values in earlier packets.

2. The firewall does not modify packets. All packets sent by the tool emerge unmodified or are dropped.

3. The load balancers only process packets addressed to them at layers 2, 3 and 4.

4. The layer-4 load balancer rewrites the destination IP and MAC addresses of packets in the `inet` → `srvr` direction, and the source addresses in the reverse direction. The tool made this inference by analyzing packets with identical payloads at the two zones of the load balancer. The tool can partially infer the header rewriting rules for even a layer-7 load balancer, by using a relaxed payload similarity metric.

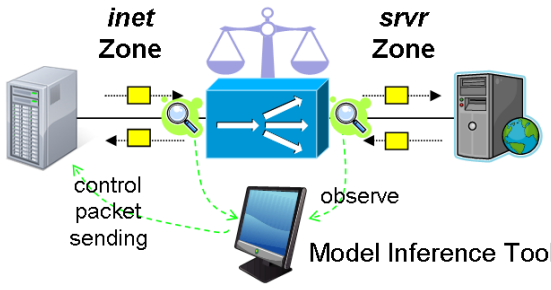

Figure 3.2. Classifier model inference tool analyzing a load balancer.

Our inference tool only offers very basic functionality. It is not fully automated. For instance, in order to avoid an exhaustive IP address search, the tool requires the load balancer's IP address and TCP port as input. Inferred packet header transformation rules and state fields only serve as a guide for further analysis, as they may not be completely

accurate. The tool cannot infer the processing rules for classifiers like SSL offload boxes that completely transform packet payloads.

Completely inferring classifier models through blackbox testing alone is not feasible. Automatic whitebox software test generation tools like DART [58] may aid model construction, if classifier source code is available. Parsing configuration manuals to produce model construction hints is another interesting research direction.

We have also prototyped a model validation tool that checks if a classifier's operations are consistent with its model. The validation tool does so by analyzing traffic traces from the different zones of a classifier. It can flag errors and incompleteness in the models themselves. It can also detect unexpected classifier behavior, as we describe next.

3.2.2 Network Planning and Troubleshooting

Our classifier model clearly and concisely describes how various classifiers interact with the network and with each other, under different network configurations. A network administrator can use this information to plan new classifier deployments and to monitor and troubleshoot existing ones.

The input pre-conditions of a classifier specify the types of packets expected by it. A network architect uses this information to plan the network topology and classifier placement so that all classifiers receive the desired packets. A network architect can use the input pre-conditions and processing rules to analyze the feasibility of placing different classifiers in sequence. For example, the firewall can be placed in front of the load balancer with little scrutiny, since the right hand sides of the firewall processing rules (Section 3.1.4) do not interfere with the conditions on the left hand sides of the load balancer processing rules(Section 3.1.4). However, placing the load balancer before the firewall may interfere with the the firewall's C_{accept} and C_{drop} clauses, as the load balancer rewrites packet headers.

Classifier processing rules indicate the packets present in different parts of a network. A network administrator can use this information to detect problems with a classifier deployment before actual network rollout. Such information also aids in troubleshooting existing

classifier deployments. In addition, it enhances automated traffic monitoring and anomaly detection.

For example, the model validation tool helped us detect a misbehaving NAT in a home network. The home NAT was not rewriting the source port numbers of packets sent by internal hosts. This behavior was automatically flagged as a violation of rules (i) and (ii) of our NAT model (Section 3.1.4). We had expected the multi-interface home NAT to use source port translation to support simultaneous TCP connections to the same destination from the same source port on multiple internal hosts. We confirmed the anomaly by opening simultaneous TCP connections from separate machines. This experience reaffirmed our model's practical utility in detecting unexpected classifier behavior.

3.2.3 Guide Networking Research

Our classifier model provides clear and concise descriptions of how various classifiers operate. Such information is very useful for networking researchers as well as companies involved in developing new network architectures. The model provides hints about how to make a new architecture compatible with existing classifiers. It also helps identify classifiers that cannot be supported.

The model influenced some key decisions in our design and implementation of the policy-aware switching layer (or *PLayer*). The *PLayer* will be presented in detail in Chapter 5. For the purpose of this discussion, the *PLayer* consists of layer-2 switches (*pswitches*) which explicitly redirect packets to the classifiers specified by administrative policy.

During our initial design phase, we did not have a load balancer model available. We assumed that a load balancer does not care about the source MAC addresses of packets received by it. We expected it to use ARP to determine the MAC address for packets in the reverse flow direction. Hence, we decided to record the last classifier traversed by a packet by rewriting its source MAC address with a *dummy* address. Contrary to our expectation, the load balancer cached the dummy source MAC addresses of packets in the forward flow

direction and used them to address packets in the reverse direction. Such packets never reached their intended destinations.

We constructed a load balancer model to help us debug this problem. The presence of the source MAC address in the interest fields of the load balancer made the problem obvious – we must not rewrite packet source MAC addresses. Hence, guided by our model, we decided to use packet encapsulation to record the previous hop of a packet, instead of rewriting source MAC addresses. Section 5.2.1 provides the details of our encapsulation scheme.

3.3 Limitations

The model presented in this thesis is only a first step towards modeling classifiers. Its three main limitations are:

1. the inability to describe highly classifier-specific operations in detail,
2. the lack of formal coverage proofs, and
3. the complexity of model specification.

Our goal of building a classifier model that can describe a wide variety of classifiers trades off specificity for generality. Our model can be extended using specific models like the Unified Firewall Model as described in Section 3.1.4, although at the expense of reducing model simplicity and conciseness. We do not capture accurate timing and causality between triggering of different processing rules, in order to keep the model simple and concise.

On the other hand, our model *may* not be general enough to represent all possible current and future classifiers. We have represented many common classifiers in our model and are not aware of any existing classifiers that cannot be represented. However, we are unable to formally prove our model's coverage.

Although models usually consist of a small number (typically < 10) of processing rules, constructing models is not a simple task, even with support from our model inference and

validation tools. We hope that models will be constructed by experts and shared through an online repository. Thus, models can be used by a wider audience who lack model construction skills.

3.4 Related Work

An axiomatic basis for communication [70] presents a general network communications model that axiomatically formulates packet forwarding, naming and addressing. This thesis presents a model tailored to represent classifier functionality and operations. The processing rules and state database in our model are similar to the forwarding primitives and local switching table in [70]. Integration of the two models may combine the practical benefits of our classifier model (*e.g.,* classifier model inference and validation tools, model repository) and the theoretical benefits of the general communications model (*e.g.,* formal validation of packet forwarding correctness through chains of classifiers).

Predicate routing [85] attempts to unify security and routing by declaratively specifying network state as a set of boolean expressions dictating the packets that can appear on various links connecting together end nodes and routers. This approach can be extended to represent a subset of our classifier model. For example, boolean expressions on the *ports* and *links* (as defined by predicate routing) of a classifier can specify the input pre-conditions of our model and indirectly hint the processing rules and transformation functions. From a different perspective, classifier models from our repository can aid the definition of the boolean expressions in a network implementing predicate routing.

[68] uses statistical rule mining to automatically group together commonly occurring flows and learn the underlying communication rules in a network. Our work has a narrower and more detailed focus on how classifiers operate. [45] describes detailed measurement techniques to evaluate production middlebox deployments.

RFC 3234 [24] presents a taxonomy of middleboxes, most of which are classifiers. Our model is more detailed than a taxonomy – it describes classifier packet processing via a

succinct and standard language. Moreover, our model induces a more fine-grained taxonomy on classifiers and middleboxes. For example, our model can represent 'classifiers that rewrite the destination IP and port number' instead of just 'classifiers operating at the transport layer'. Our model does not capture the middlebox failover modes and functional versus optimizing roles identified by RFC 3234.

The Unified Firewall Model [79] and IETF BEHAVE [5] working group characterize the functionality and behavior of specific classifiers and middleboxes – firewalls and NATs in this case. Our generic model enables us to compare different classifiers and study their interactions. Our generic models can be enhanced by plugging in specific models.

3.5 Summary

In this chapter, we presented a simple classifier model and illustrated how various commonly used classifiers can be described by it. The model guides classifier-related research and aids classifier deployments. Our work is only an initial step in this direction and calls for the support of the classifier research and user communities to further refine the model and to contribute model instances for the many different kinds of classifiers that exist today.

Chapter 4

Packet Classification as a Fundamental Primitive

As described in Chapter 2, packet classification in current networks suffers from three main limitations – configuration complexity, inflexibility, and inefficiency. We argue that the root cause of these limitations is that packet classification is implemented and deployed in an ad-hoc manner, in spite of its ubiquity in today's networks.

In this chapter, we advocate that packet classification should be a *fundamental primitive* in the network stack. Towards this goal, we define a new *classification layer* (or *CLayer*) in the protocol stack, and two associated control plane protocols – *policy-based classifier deployment* and *generic classification offload*. *Policy-based classifier* deployment enables network administrators to explicitly deploy special-purpose classifiers in the sequence they desire. *Generic classification offload* enables classifiers and other entities in the network to negotiate and share the classification load for different applications amongst themselves. A packet's *CLayer* header carries information that is used by these two control plane protocols.

Our proposal addresses the configuration complexity, inflexibility and inefficiency of current classification solutions, as follows:

- In **Policy-based classifier deployment**, network administrators configure the sequence of classifiers to be traversed by different kinds of traffic by simply specifying *policies* at a centralized policy controller. The underlying forwarding mechanisms explicitly redirect traffic to the required classifiers. This way, these mechanisms guarantee that traffic will traverse the required sequence of classifiers under all network churn conditions. The centralized point of control simplifies configuration and enhances flexibility. A network administrator is no longer forced to rely on ad-hoc and inflexible mechanisms like physical re-wiring and tweaking spanning tree link weights to change policies. Explicit redirection improves network efficiency by utilizing all available classifiers, and by sending traffic only through necessary ones.

- **Generic classification offload** enables network administrators to implement classification offload for multiple classification applications using a single mechanism. This simplifies network configuration by avoiding the hassle of separately implementing and configuring diverse mechanisms at different protocol layers. Generic classification offload improves network efficiency by enabling explicit co-ordination between different network entities, and by leveraging *CLayer* headers to avoid deep packet inspection. Its generic nature allows it to provide classification offload to new applications, and thereby enhances network flexibility.

In the rest of this chapter, we first present the *CLayer*, and then provide an overview of *policy-based classifier deployment* and *generic classification offload*. Chapters 5 and 6 describe the ideas introduced here in more detail.

4.1 Classification Layer

The main contribution of this thesis is a new *classification layer* in the network stack. The classification layer, in short the *CLayer*, provides a well-defined and easily accessible location within a packet to carry the signaling messages of policy-based classifier deployment and generic classification offload.

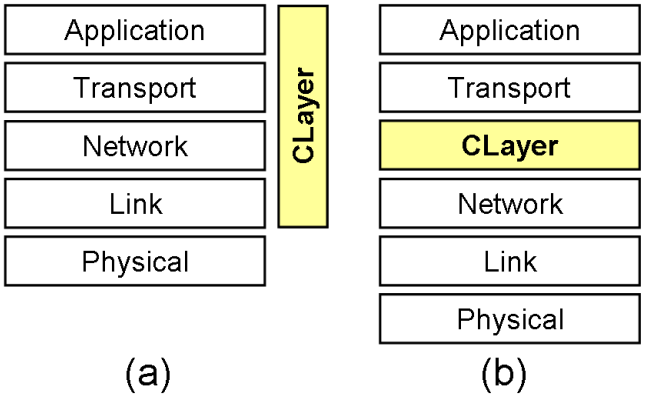

Figure 4.1. (a) Logical *CLayer* location in protocol stack, (b) A practical implementation choice.

The *CLayer* logically spans the link layer to the application layer, as shown in Figure 4.1(a). This enables it to simultaneously support classification applications at different layers and those that span multiple layers. Respecting the wide prevalence of packet classification in today's networks, we advocate a separate layer for semantic clarity.

In practice, we advocate implementing the *CLayer* as a new layer between the network and transport layers, as shown in Figure 4.1(b). In order to avoid extensive changes to current forwarding infrastructures, the *CLayer* should be implemented at or above the network layer. If the *CLayer* is below the network layer, routers must be upgraded to understand *CLayer* headers, at least to the extent that they can be safely ignored. Network layering guidelines dictate that a protocol layer should only use the services of layers below itself. To minimize violation of this architectural guideline, we place the *CLayer* immediately above the network layer, *i.e.,* below the transport layer. Another option is to implement the *CLayer* as a sub-layer within the network layer; for instance, as an IP option. However, implementing the *CLayer* at or above the network layer still violates the layering guideline for layer-2 only applications.

Implementing the *CLayer* at or above the network layer ensures that *CLayer* headers are preserved end-to-end in the absence of layer-7 'proxies' on the path. A layer-7 proxy (for example, layer-7 load balancer) acts as a TCP endpoint for the client connection and

opens a new TCP connection to its final destination. We assume that such layer-7 proxies are *CLayer*-aware enough at least to blindly forward along any *CLayer* headers they receive.

A standardized location of the *CLayer* within a packet header provides easy access to the classification information contained in it. In Section 2.3.3, we saw that a web load balancer gratuitously implements complex layer-7 processing in order to read HTTP cookie values. When using the *CLayer*, the load balancer avoids such layer-7 processing overheads by simply reading the cookie value from the packet's *CLayer* header.

Ideally, the *CLayer* header has a flexible format consisting of key-value pairs. For practical high-speed implementations, a more rigid fixed-field header structure may be used. We provide more details about the header structure in Chapter 6 where we describe *GOff*, an instantiation of the generic classification offload protocol.

4.2 Policy-based Classifier Deployment

In Chapter 2, we saw that current classifier deployment solutions implicitly place classifiers on the network path traversed by traffic. This introduces a tight coupling between the network's logical topology, *i.e.,* classifier traversal sequences for different traffic types, and the physical topology. Network administrators indirectly tweak network path selection mechanisms like spanning tree construction, routing and VLANs to coerce traffic to flow through paths containing the desired classifiers. Such ad-hoc mechanisms are hard to configure, cannot guarantee correct traversal, and are inefficient and inflexible.

Policy-based classifier deployment is a classifier deployment solution that avoids the problems of current mechanisms. It is based on the following two design principles:

1. Take special-purpose classifiers off the network path.

   Special-purpose classifiers should not be placed on the physical network paths as shown in Figure 4.2(a). Instead, they are plugged into routers and switches just like regular endhosts, as shown in Figure 4.2(b).

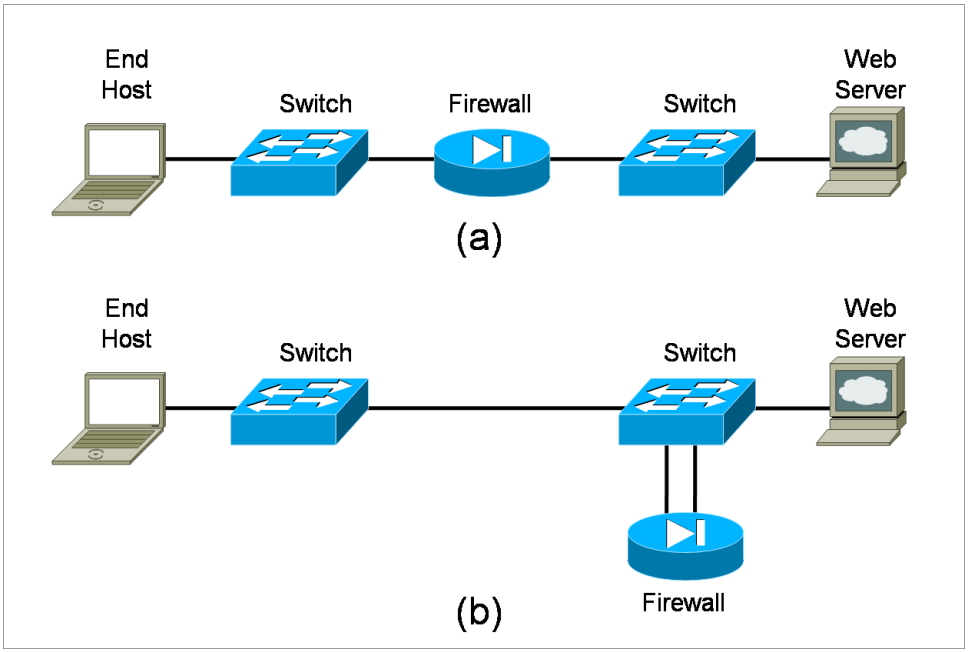

Figure 4.2. (a) Firewall deployed on network path, (b) Firewall deployed off path.

2. Separate policy from reachability.

   The sequence of classifiers traversed by different types of traffic should not be implicitly dictated by the physical network topology or the interactions between unrelated network configuration knobs like forwarding protocols, VLAN configuration and traffic engineering. Instead, network administrators explicitly specify the classifier sequence for different traffic types, which must then be explicitly enforced by the network.

We provide an overview of policy-based classifier deployment using the simple example in Figure 4.3. Endhost A and web server W are plugged into switch S. Firewall F is also plugged into switch S, just like A and W. The network administrator specifies the classifier traversal policies for different traffic types at the centralized policy controller C. In this basic example, the policy simply states that all HTTP traffic to W should traverse a firewall. The policy controller disseminates this policy to switch S. When switch S receives a packet from A, it looks up the matching policy from its policy database. As specified by the policy, S explicitly forwards the packet to firewall F. If F does not drop the packet, it arrives back at S. S again looks up the matching policy and determines that the packet

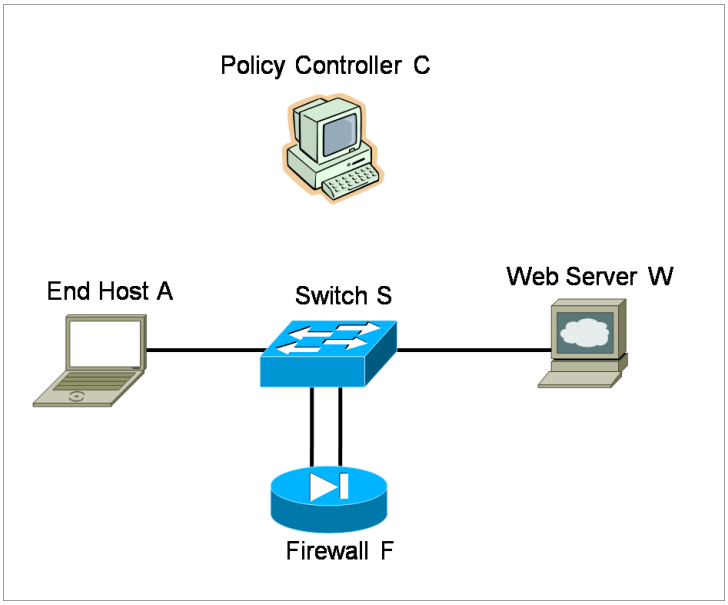

Figure 4.3. Basic packet forwarding in a policy-based deployment.

has traversed all required classifiers. Hence, it explicitly forwards the packet to its final destination, W.

In this example, switch S looked up the policy for a packet twice – first on arrival from A and second on arrival from F. Switch S can avoid the second policy lookup by recording the sequence of classifiers that must process the packet in its *CLayer* header. Each classifier that processes the packet advances a pointer in the *CLayer* header to point to the next hop. A switch receiving the packet can simply use this information to identify a packet's next hop and forward it there.

In this simple example, we had only one switch and one classifier. In practice, there are multiple switches and routers, many different classifiers, multiple instances of each classifier type, and a larger number of policies. The switches and routers correctly redirect packets to all the classifiers specified by policy. They also load balance traffic across different instances of the same classifier. We describe these scenarios in detail when we present the policy-aware switching layer in Chapter 5.

Separating policy from reachability and centralized control of networks have been proposed in previous work [60, 51]. Explicitly redirecting network packets to pass through

off-path middleboxes is based on the well-known principle of indirection [88, 94, 59]. We combine these two general principles to revise the ad-hoc manner in which classifiers are deployed today. We discuss related work in more detail in Section 5.8.

4.3 Generic Classification Offload

As mentioned in Chapter 2, classification offload improves the scalability of classification. Currently, there exists no standard mechanisms to implement classification offload. Various point solutions tailored to specific applications and protocol layers exist. These solutions are inefficient and inflexible to support new applications.

Generic Classification Offload enables classification offload across different applications and protocol layers. By avoiding the need to implement and configure separate offload mechanisms, it simplifies network configuration. Generic classification offload efficiently uses *CLayer* headers to carry signaling messages, and flexibly supports new applications.

The main idea behind Generic Classification Offload is to enable explicit coordination between two types of entities in the network – *classifiers* and *helpers*. *Classifiers* classify a packet and take action depending on the results of the classification. Examples of classifiers are load balancers, firewalls, and routers. *Helpers* aid *classifiers* by performing classification tasks on their behalf. Examples of *helpers* are endhosts that provide HTTP cookies to web load balancers to aid them in session identification, and edge routers in a Diffserv domain which set code points in packet headers for use by core routers. Note that in the latter example, an edge router is both a *helper* and a *classifier*, as it also classifies packets to determine their next hop, in addition to helping core routers.

Helpers and *classifiers* coordinate using a signaling protocol. *Helpers* advertise their classification capabilities to *classifiers*. *Classifiers* request classification support from *helpers*. An upstream *helper* classifies a packet and embeds the 'results' of classification for use by a downstream *classifier*. A classification result is an opaque bag of bits that is interpreted by the *classifier* to which it is addressed. The *classifier* reads the results and appropriately

processes the packet. The signaling messages and classification results are all embedded inside a packet's *CLayer* headers.

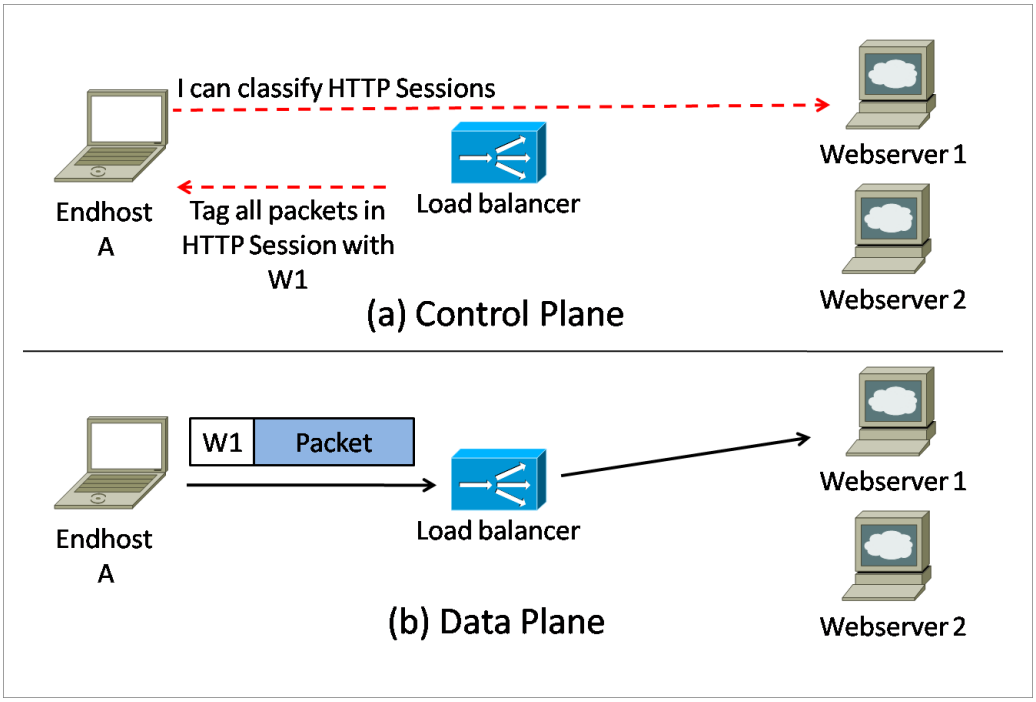

Figure 4.4. Basic operations in generic classification offload.

To illustrate the basic functionality of generic classification offload, consider the example in Figure 4.4. Server 1 and 2 each host an online shopping application. Clients communicate with the load balancer L. L spreads the load across the two servers while maintaining HTTP session 'stickiness', *i.e.,* all HTTP connections in the same HTTP session are forwarded to the same server instance. Client A can easily identify the HTTP session of the different HTTP connections originated by it. It advertises this ability when it first communicates with L, as illustrated in Figure 4.4(a). On receiving the advertisement, L requests A to tag all packets in the same HTTP session with the label $W1$ denoting server 1. A adds the tag to the *CLayer* headers of all packets in the same session, as requested. L easily reads this tag without layer-7 processing, and forwards the packet to the correct server instance.

In the above example, there was only one *helper* and one *classifier*. In practice, there will be multiple helpers and classifiers on the path of a packet. We present such complex

scenarios in more detail in Chapter 6 where we describe *GOff*, a robust signaling protocol implementation for Generic Classification Offload.

Our proposal borrows ideas from label switching [25], Diffserv [2], IPv6 flowid [19], HTTP cookies [15] and ECN [38]. Our main contribution here is the generic offload mechanism that works across different applications. We describe related work in Section 6.5.

4.4 Summary

In this chapter, we introduced a new classification layer in the network stack, and provided an overview of its control plane protocols – policy-based classifier deployment and generic classification offload. In the next two chapters, we describe the policy-aware switching layer and *GOff*, two specific instantiations of these control plane protocols.

Chapter 5

Policy-aware Switching Layer

The previous chapter presented an overview of *policy-based classifier deployment*. In this chapter, we describe it in greater detail using the *policy-aware switching layer*. The policy-aware switching layer, or *PLayer* in short, is a practical instantiation of policy-based classifier deployment targeted at data centers and enterprise networks.

In recent years, data centers have rapidly grown to become an integral part of the Internet fabric [22]. These data centers typically host tens or even thousands of different applications [46], ranging from simple web servers providing static content to complex e-commerce applications. To protect, manage and improve the performance of these applications, data centers deploy a large variety of special-purpose *classifiers* such as firewalls, load balancers, SSL offloaders and intrusion prevention boxes, and *middleboxes* such as web caches.

As we have already seen in Chapter 2, current classifier deployment mechanisms in data centers are inflexible, inefficient and hard to configure. The *PLayer* is a new layer-2 for data center networks that addresses these limitations. To ease deployment in existing data centers, the *PLayer* aims to support existing classifiers, application servers and external clients without any modifications, and to minimize changes required in switches.

In brief, the *PLayer* consists of policy-aware switches, or *pswitches*, which maintain

the classifier traversal requirements of all applications in the form of *policy specifications*. *Pswitches* explicitly redirect incoming traffic to the required classifiers. This explicit redirection guarantees classifier traversal in the policy-mandated sequence. The low-latency links in a typical data center network enable off-path placement of classifiers with minimal performance overheads. Off-path classifier placement simplifies topology modifications and enables efficient usage of existing classifiers. For example, adding an SSL offload box in front of HTTPS traffic simply involves plugging in the SSL offload box into a *pswitch* and configuring the appropriate HTTPS traffic policy at a centralized policy controller. The system automatically ensures that the SSL box is only traversed by HTTPS traffic while the firewall and the load balancer are shared with HTTP traffic.

Keeping existing classifiers and servers unmodified, supporting classifiers that modify frames, and guaranteeing correct classifier traversal under all conditions of policy, classifier and network churn make the design and implementation of the *PLayer* a challenging problem. We have prototyped *pswitches* in software using Click [72] and evaluated its functionality on a small testbed.

The rest of this chapter is organized as follows. The next section provides an overview of the *PLayer* design and its associated challenges. Sections 5.2 to 5.4 present the details of how our solution addresses these challenges. Section 5.5 presents our implementation and evaluation results. Section 5.6 analyzes *PLayer* operations using a formal model. Section 5.7 lists the limitations of the *PLayer*, and Section 5.8 describes related work. We conclude this chapter with a brief discussion of clean slate and stateful designs in Section 5.9. Appendix A.1 presents detailed algorithms explaining how *pswitches* process frames.

5.1 Design Overview

The policy-aware switching layer (*PLayer*) adheres to the two design principles of policy-based classifier deployment, introduced in Section 4.2:

1. *Separate policy from reachability.*

The sequence of classifiers traversed by application traffic is explicitly dictated by data center policy, and not implicitly by network path selection mechanisms like layer-2 spanning tree construction and layer-3 routing.

2. *Take classifiers off the physical network path.*

   Rather than placing classifiers on the physical network path at choke points in the network, classifiers are plugged in off the physical network data path and traffic is explicitly forwarded to them.

Explicitly redirecting traffic through off-path classifiers is based on the well-known principle of indirection [88, 94, 59]. A data center network is a more apt environment for indirection than the wide area Internet due to its very low inter-node latencies.

The *PLayer* consists of enhanced layer-2 switches called policy-aware switches or *pswitches*. Unmodified classifiers are plugged into a *pswitch* just like servers are plugged into a regular layer-2 switch. However, unlike regular layer-2 switches, *pswitches* forward frames according to the policies specified by the network administrator.

Policies define the sequence of classifiers to be traversed by different traffic types. A policy is of the form: *[Start Location, Traffic Selector]→Sequence*. The left hand side defines the applicable traffic – frames with 5-tuples (*i.e.,* source and destination IP addresses and port numbers, and protocol type) matching the *Traffic Selector* arriving from the *Start Location*. The right hand side specifies the sequence of classifier types (not instances) to be traversed by this traffic [1] . We use *frame 5-tuple* to refer to the 5-tuple of the packet within the frame.

Policies are automatically translated by the *PLayer* into *rules* that are stored at *pswitches* in rule tables. A rule is of the form *[Previous Hop, Traffic Selector] : Next Hop*. Each rule determines the classifier or server to which traffic of a particular type, arriving from the specified previous hop, should be forwarded next. Upon receiving a frame,

[1] Classifier interface information can also be incorporated into a policy. For example, frames from an external client to an internal server must enter a firewall via its *red* interface, while frames in the reverse direction should enter through the *green* interface.

the *pswitch* matches it to a rule in its table, if any, and then forwards it to the next hop specified by the matching rule.

The *PLayer* relies on centralized *policy* and *classifier* controllers to set up and maintain the rule tables at the various *pswitches*. Network administrators specify policies at the policy controller, which then reliably disseminates them to each *pswitch*. The centralized classifier controller monitors the liveness of classifiers and informs *pswitches* about the addition or failure of classifiers.

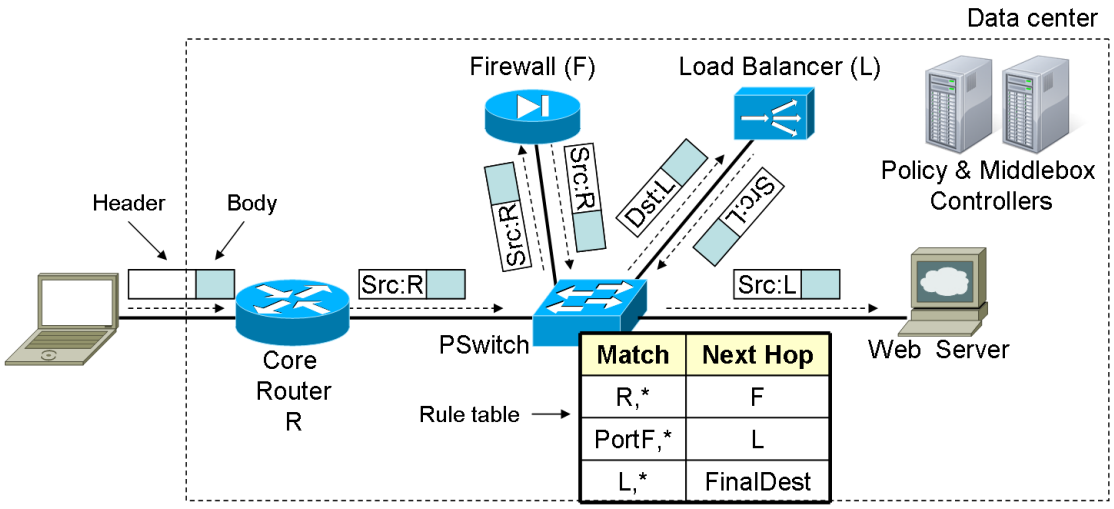

Figure 5.1. A simple *PLayer* consisting of only one *pswitch*.

To better understand how the *PLayer* works, we next present three examples of increasing complexity that demonstrate its key functionality. In practice, the *PLayer* consists of multiple *pswitches* inter-connected together in complex topologies. For example, in a typical data center topology (Section 2.1.1), *pswitches* would replace layer-2 switches at the aggregation layer. However, for ease of exposition, we start with a simple example containing only a single *pswitch*.

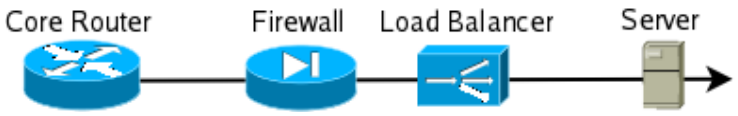

Figure 5.2. A simplified snippet of the data center topology in Figure 2.2, highlighting the on-path classifier placement.

5.1.1 Single *Pswitch*

Figure 5.1 shows how the *PLayer* implements the policy induced by the physical topology in Figure 5.2, where all frames entering the data center are required to traverse a firewall and then a load balancer before reaching the servers.

When the *pswitch* receives a frame, it performs the following three operations:

1. Identify the previous hop traversed by the frame.
2. Determine the next hop to be traversed by the frame.
3. Forward the frame to its next hop.

The *pswitch* identifies frames arriving from the core router and the load balancer based on their source MAC addresses (R and L, respectively). Since the firewall does not modify the MAC addresses of frames passing through it, the *pswitch* identifies frames coming from it based on the ingress interface (*IfaceF*) they arrive on. The *pswitch* determines the next hop for the frame by matching its previous hop information and 5-tuple against the rules in the rule table. In this example, the policy translates into the following three rules:

1. $[R, *] : F$
2. $[IfaceF, *] : L$
3. $[L, *] : FinalDest$

The first rule specifies that every frame entering the data center (*i.e.,* every frame arriving from core router R) should be forwarded to the firewall (F). The second rule specifies that every frame arriving from the firewall should be forwarded to the load balancer (L). The

third rule specifies that frames arriving from the load balancer should be sent to the final destination, *i.e.,* the server identified by the frame's destination MAC address. The *pswitch* forwards the frame to the next hop determined by the matching rule, encapsulated in a frame explicitly addressed to the next hop. It is easy to see that the *pswitch* correctly implements the original policy through these rules, *i.e.,* every incoming frame traverses the firewall followed by the load balancer.

5.1.2 Multiple Classifier Instances

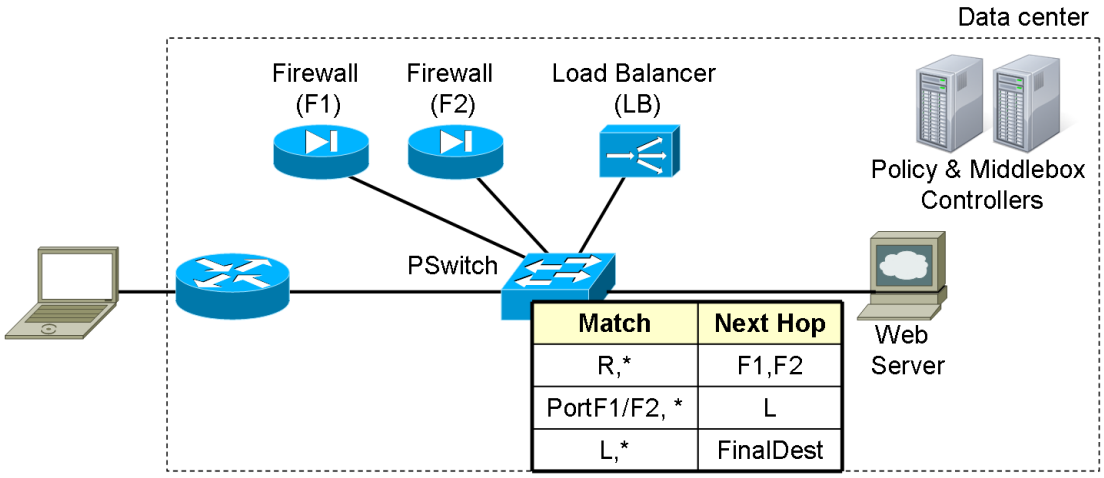

Figure 5.3. Load balancing traffic across two equivalent classifier instances.

Multiple equivalent instances of classifiers are often deployed for scalability and fault-tolerance. Figure 5.3 shows how the *PLayer* can load balance incoming traffic across two equivalent firewalls, $F1$ and $F2$. The first rule in the table specifies that incoming frames can be sent either to firewall $F1$ or to firewall $F2$. Since the firewall maintains per-flow state, the *pswitch* uses a flow- direction-agnostic consistent hash on a frame's 5-tuple to select the same firewall instance for all frames in both forward and reverse directions of a flow.

5.1.3 Multiple Policies and *Pswitches*

The more complex example in Figure 5.4 illustrates how the *PLayer* supports different policies for different applications and how forwarding load is spread across multiple *pswitches*. Web traffic has the same policy as before, while Enterprise Resource Planning (ERP) traffic is to be scrubbed by a dedicated custom firewall (W) followed by an Intrusion Prevention Box (IPB). The classifiers are distributed across the two *pswitches* A and B. The rule table at each *pswitch* has rules that match frames coming from the entities connected to it. For example, rules at *pswitch* A match frames coming from classifiers $F1$ and L, and the core router R. For sake of simplicity, we assume that all frames with TCP port 80 are part of web traffic and all others are part of ERP traffic.

A frame (say, an ERP frame) entering the data center first reaches *pswitch* A. *Pswitch* A looks up the most specific rule for the frame ($[R, *] : W$) and forwards it to the next hop (W). The *PLayer* uses existing layer-2 mechanisms (*e.g.,* spanning tree based Ethernet forwarding) to forward the frame to its next hop, instead of inventing a new forwarding mechanism. *Pswitch* B receives the frame after it is processed by W. It looks up the most specific rule from its rule table ($[\mathit{Iface}\,W, *] : IPB$) and forwards the frame to the next hop (IPB). An HTTP frame entering the data center matches different rules and thus follows a different path.

5.1.4 Discussion

The three examples discussed in this section provide a high level illustration of how the *PLayer* achieves the three desirable properties of correctness, flexibility and efficiency. The explicit separation between policy and the physical network topology simplifies configuration. The desired logical topologies can be easily implemented by specifying appropriate policies at the centralized policy controller, without tweaking spanning tree link costs and IP gateway settings distributed across various switches and servers. By explicitly redirecting frames only through the classifiers specified by policy, the *PLayer* guarantees that classifiers are neither skipped nor unnecessarily traversed. Placing classifiers off the physical

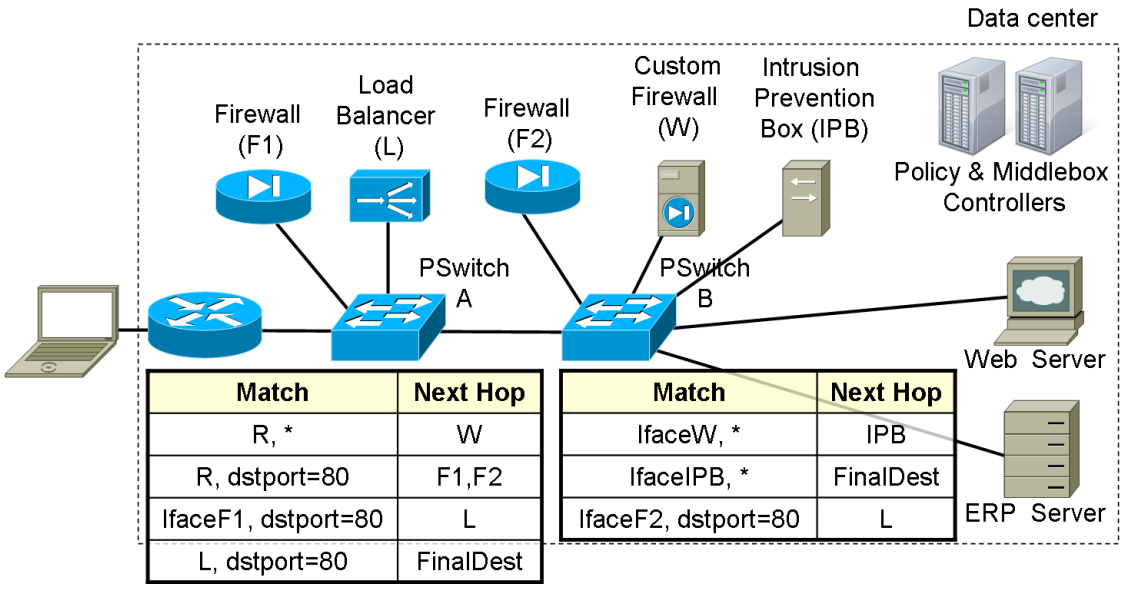

Figure 5.4. Different policies for web and ERP applications.

network path prevents large scale traffic shifts on classifier failures and ensures that classifier resources are not wasted serving unnecessary traffic or get stuck on inactive network paths.

The *PLayer* operates at layer-2 since data centers are pre-dominantly layer-2 [46]. It re-uses existing tried and tested layer-2 mechanisms to forward packets between two points in the network rather than inventing a custom forwarding mechanism. Furthermore, since classifiers like firewalls are often not explicitly addressable, the *PLayer* relies on simple layer-2 mechanisms described in Section 5.2.2 to forward frames to these classifiers, rather than more heavy-weight layer-3 or higher mechanisms.

In the next three sections, we discuss how the *PLayer* addresses the three main challenges listed below:

- **(i) Minimal Infrastructure Changes:** Support existing classifiers and servers without any modifications and minimize changes to network infrastructure like switches.
- **(ii) Non-transparent Classifiers :** Handle classifiers that modify frames while

specifying policies and while ensuring that all frames in both forward and reverse directions of a flow traverse the same classifier instances.

- **(iii) Correct Traversal Under Churn :** Guarantee correct classifier traversal during classifier churn and conflicting policy updates.

5.2 Minimal Infrastructure Changes

Minimizing changes to existing network forwarding infrastructure and supporting unmodified classifiers and servers is crucial for *PLayer* adoption in current data centers. In this section, we describe how we meet this challenge. In addition, we explain a *pswitch*'s internal structure and operations, and thus set the stage for describing how we solve other challenges in subsequent sections.

5.2.1 Forwarding Infrastructure

The modular design of *pswitches*, reliance on standard data center path selection mechanisms to forward frames, and encapsulation of forwarded frames in new Ethernet-II frames help meet the challenge of minimizing changes to the existing data center network forwarding infrastructure.

Pswitch Design & Standard Forwarding

Figure 5.5 shows the internal structure of a *pswitch* with N interfaces. For ease of explanation, each physical interface is shown to comprise of two separate logical interfaces – an input interface and an output interface. A *pswitch* consists of two independent parts – the Switch Core and the Policy Core, described below:

1. *Switch Core*

   The Switch Core provides the forwarding functionality of a standard Ethernet switch. It forwards Ethernet frames received at its interfaces based on their destination MAC

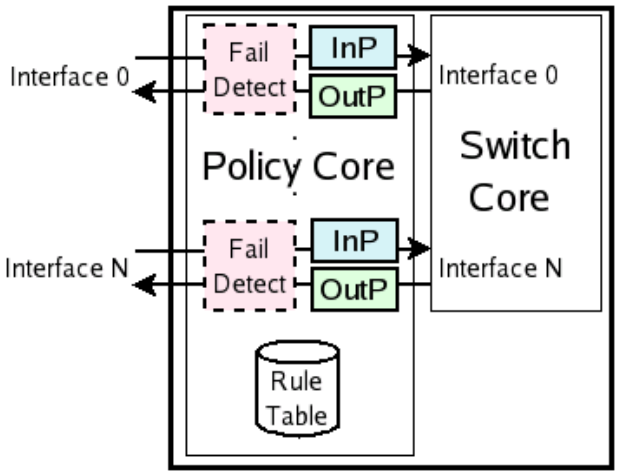

Figure 5.5. Internal components of a *pswitch*.

addresses. If the destination MAC address of a frame received at an interface, say X, was previously learned by the Switch Core, then the frame is forwarded only on the interface associated with the learned MAC address. Else, the frame is flooded on all Switch Core interfaces other than X. The Switch Core coordinates with Switch Cores in other *pswitches* through existing protocols like the Spanning Tree Protocol [10] to construct a loop-free forwarding topology.

2. *Policy Core*

The Policy Core redirects frames [2] to the classifiers dictated by policy. It consists of multiple modules: The RuleTable stores the rules used for matching and forwarding frames. Each *pswitch* interface has an inP, an outP and a FailDetect module associated with it. An inP module processes a frame as it enters a *pswitch* interface – it identifies the frame's previous hop, looks up the matching rule and emits it out to the corresponding Switch Core interface for regular forwarding to the next hop specified by the rule. An outP module processes a frame as it exits a *pswitch* interface, decapsulating or dropping it as explained later in the section. The FailDetect module of a *pswitch* interface monitors the liveness of the connected classifier (if any) using standard mechanisms like ICMP pings, layer-7 content snooping, SNMP polling,

[2]Only frames containing IP packets are considered. Non-IP frames like ARP requests are forwarded by the Switch Core as in regular Ethernet switches.

and TCP health checks. It reports the collected status information to the classifier controller.

The Switch Core appears like a regular Ethernet switch to the Policy Core, while the Policy Core appears like a multi-interface device to the Switch Core. This clean separation allows us to re-use existing Ethernet switch functionality in constructing a *pswitch* with minimal changes, thus simplifying deployment. The Switch Core can also be easily replaced with an existing non-Ethernet forwarding mechanism, if required by the existing data center network infrastructure.

Encapsulation

A frame redirected by the Policy Core is encapsulated in a new Ethernet-II frame, identified by a new *EtherType* code from the IEEE EtherType Field Registration Authority [17], as shown in Figure 5.6. The outer frame's destination MAC address is set to that of the next hop classifier or server, and the source MAC is set to that of the original frame (or of the last classifier instance traversed, if any) in order to enable MAC address learning by Switch Cores. An encapsulated frame also includes a 1-byte *Info* field that tracks the version number of the policy used to redirect it.

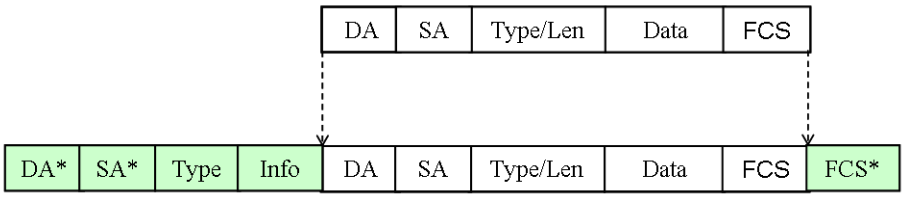

Figure 5.6. Cisco ISL [20] style frame encapsulation.

We encapsulate, rather than overwrite the original frame headers, as preserving the MAC addresses of the original frame is often required for correctness. For example, firewalls may filter based on source MAC addresses , and load-balancers set the destination MAC address to that of a server chosen based on dynamic load conditions. Although the 15-byte

encapsulation overhead may increase frame size beyond the 1500 byte MTU, an encapsulated frame is below the size limit accepted by most layer-2 switches. For example, Cisco switches allow 1600 byte 'baby giants'.

Incremental Deployment

Incorporating the *PLayer* into an existing data center does not require a fork-lift upgrade of the entire network. Only switches which connect to the external network and those into which servers requiring classifier traversal guarantees are plugged in, need to be converted to *pswitches*. Other switches need not be converted if they can be configured or modified to treat encapsulated frames with the new *EtherType* as regular Ethernet frames. Classifiers can also be plugged into a regular switch. However, transparent classifiers must be accompanied by the inline SrcMacRewriter device (described in Section 5.2.2). If the data center contains backup switches and redundant paths, *pswitches* can be smoothly introduced without network downtime by first converting the backup switches to *pswitches*.

5.2.2 Unmodified Classifiers and Servers

Pswitches address the challenge of supporting unmodified classifiers and servers in three ways:

1. Ensure that only relevant frames in standard Ethernet format reach classifiers and servers.
2. Use only non-intrusive techniques to identify a frame's previous hop.
3. Support varied classifier addressing requirements.

Frames reaching Classifiers and Servers

The outP module of a *pswitch* interface directly connected to a classifier or server emits out a unicast frame only if it is MAC addressed to the connected classifier or server.

Dropping other frames, which may have reached the *pswitch* through standard Ethernet broadcast forwarding, avoids undesirable classifier behavior (*e.g.,* a firewall can terminate a flow by sending TCP RSTs if it receives an unexpected frame). The OUTP module also decapsulates the frames it emits and thus the classifier or server receives standard Ethernet frames it can understand.

Previous Hop Identification

A *pswitch* does not rely on explicit classifier support or modifications for identifying a frame's previous hop. The previous hop of a frame can be identified in three possible ways:

1. source MAC address if the previous hop is a classifier that changes the source MAC address,
2. *pswitch* interface on which the frame arrives if the classifier is directly attached to the *pswitch*, or
3. VLAN tag if the data center network has been divided into different functional zones using VLANs (*i.e.,* external web servers, firewalls, etc.).

If none of the above three conditions hold (for example, in a partial *pswitch* deployment where classifiers are plugged into regular Ethernet switches), then we install a simple stateless in-line device, SRCMACREWRITER, in between the classifier and the regular Ethernet switch to which it is connected. SRCMACREWRITER inserts a special source MAC address that can uniquely identify the classifier into frames emitted by the classifier, as in option 1 above.

The previous hop identification options described above make the following two assumptions:

1. Classifiers and servers of interest are all part of the same layer-2 network, as in common data center deployments today. Classifiers in a different layer-2 network cannot be identified as the connecting routers overwrite the source MAC address of frames.

2. The data center network is secure enough to prevent source MAC address and VLAN spoofing.

Classifier Addressing

Many classifiers like firewalls transparently operate inline with traffic and do not require traffic to be explicitly addressed to them at layer-2 or layer-3. Moreover, for many such classifiers, traffic *cannot* be explicitly addressed to them, as they lack a MAC address. We solve this problem by assigning a fake MAC address to such a classifier instance when it is registered with the classifier controller. The fake MAC address is used as the destination MAC of encapsulated frames forwarded to it. If the classifier is directly connected to a *pswitch*, the *pswitch* also fills in this MAC address in the source MAC field of encapsulated frames forwarded to the next hop. If it is not directly attached to a *pswitch*, this MAC address is used by the SrcMacRewriter element described in the previous section. In all cases, the classifier remains unmodified.

In contrast, some classifiers like load balancers often require traffic to be explicitly addressed to them at layer-2, layer-3 or both. The characteristics of each classifier type are obtained from technical specifications or from our classifier model (Chapter 3). We support classifiers that require layer-3 addressing using per-segment policies to be described in Section 5.3. We support classifiers that require layer-2 addressing by having the outP module rewrite the destination MAC address of a frame to the required value before emitting it out to such a classifier.

5.3 Non-Transparent Classifiers

Non-transparent classifiers, *i.e.,* classifiers that modify frame headers or content (for *e.g.,* load balancers), make end-to-end policy specification and consistent classifier instance selection challenging. By using per-segment policies, we support non-transparent classifiers in policy specification. By enhancing policy specifications with hints that indicate which

frame header fields are left untouched by non-transparent classifiers, we enable the classifier instance selection mechanism at a *pswitch* to select the same classifier instances for all packets in both forward and reverse directions of a flow, as required by stateful classifiers like firewalls and load balancers.

Classifiers may modify frames reaching them in different ways. MAC-address modification aids previous hop identification but does not affect traffic classification or classifier instance selection since they are independent of layer-2 headers. Similarly, payload modification does not affect policy specification or classifier instance selection, unless deep packet inspection is used for traffic classification. Traffic classification and flow identification mainly rely on a frame's 5-tuple. Classifiers that fragment frames do not affect policy specification or classifier instance selection as long as the frame 5-tuple is the same for all fragments. In the remainder of this section, we describe how we support classifiers that modify frame 5-tuples. We also provide the details of our basic classifier instance selection mechanism in order to provide the context for how non-transparent classifiers and classifier churn (Section 5.4.3) affect it.

5.3.1 Policy Specification

Classifiers that modify frame 5-tuples are supported in policy specification by using *per-segment policies*. We define the bi-directional end-to-end traffic between two nodes, *e.g., A* and *B*, as a *flow*. Figure 5.7 depicts a flow passing through a firewall unmodified, and then a load balancer that rewrites the destination IP address IP_B to the address IP_W of an available web server. Frame modifications by the load balancer preclude the use of a single concise *Selector*.

Per-segment policies 1 and 2 shown in Figure 5.7 together define the complete policy. Each per-segment policy matches frames during a portion of its end-to-end flow. Per-segment policies also enable the definition of policies that include classifiers which require traffic to be explicitly addressed to them at the IP layer.

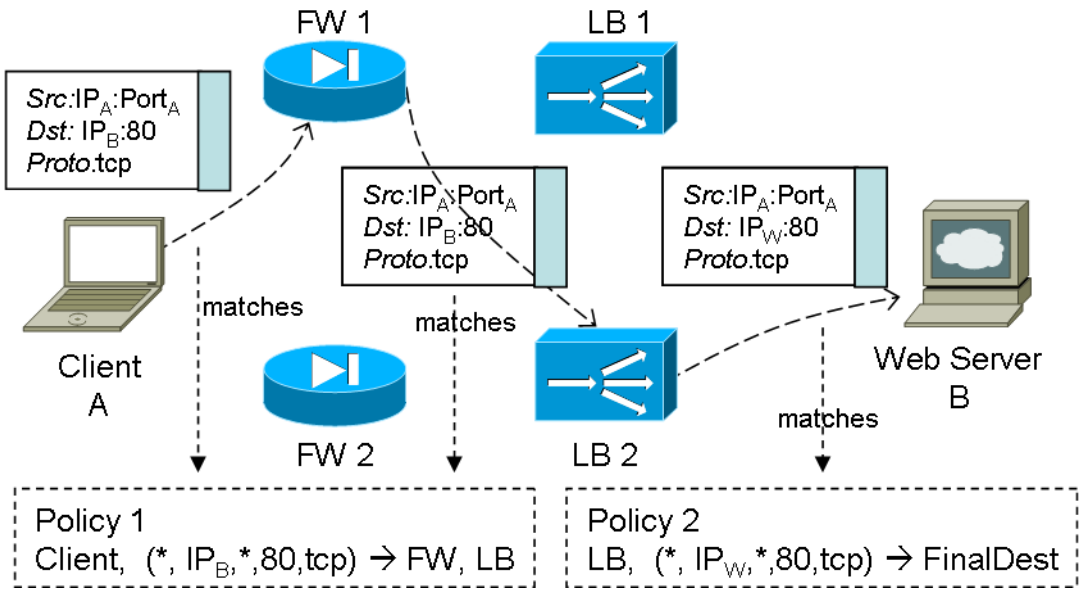

Figure 5.7. Policies for different segments of the logical classifier sequence traversed by traffic between A and B.

5.3.2 Classifier Instance Selection

The basic classifier instance selection mechanism uses consistent hashing to select the same classifier instance for all frames in both forward and reverse directions of a flow. A frame's 5-tuple identifies the flow to which it belongs. A hash value h is calculated over the frame's 5-tuple, taking care to ensure that it is flow direction agnostic, *i.e.,* source and destination fields in the 5-tuple are not distinguished in the calculation of h. The ids [3] of all live instances of the specified classifier type are arranged in a ring as shown in Figure 5.8, and the instance whose id is closest to h in the counter-clockwise direction is selected [89].

5.3.3 Policy Hints for Classifier Instance Selection

We first consider the case where classifiers do not change all the fields of the 5-tuple. Based on classifier semantics and functionality, network administrators indicate the frame 5-tuple fields to be used in classifier instance selection along with the policy. For classifiers

[3]Classifier instance ids are randomly assigned by the classifier controller when the network administrator registers the instance.

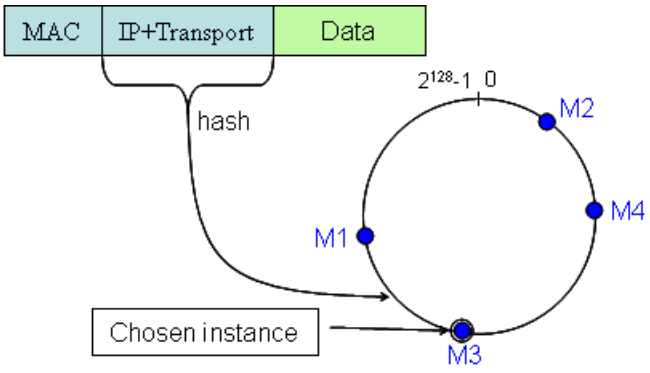

Figure 5.8. Choosing a classifier instance for a flow from among four instances $M1 - M4$ using consistent hashing.

that do not modify frames, the entire frame 5-tuple is used to identify a flow and select the classifier instance for it, as described in the previous section.

When classifiers modify the frame 5-tuple, instance selection can no longer be based on the entire 5-tuple. For example, in the $A \rightarrow B$ flow direction in Figure 5.7, the load balancer instance is selected when the frame 5-tuple is $(IP_A, IP_B, Port_A, Port_B, tcp)$. On the $B \rightarrow A$ reverse direction, the load balancer instance is to be selected when the frame 5-tuple is $(IP_W, IP_A, Port_B, Port_A, tcp)$. The policy hints that a load balancer instance should be selected only based on frame 5-tuple fields unmodified by the load balancer, *viz.*, IP_A, $Port_A$, $Port_B$ and tcp (although source and destination fields are interchanged).

Next, we consider the case where a classifier changes all the fields of the 5-tuple. Here, we assume that the classifier always changes the frame's source IP address to its own IP address, so that regular layer-3 routing can be used to ensure that reverse traffic reaches the same classifier instance. In practice, we are not aware of any classifiers that violate this assumption. However, for the sake of completeness, we discuss below how *pswitches* can be enhanced with per-flow state to support these classifiers, if they exist.

Stateful *Pswitches*

A regular *pswitch*, *i.e.,* a *stateless pswitch*, is enhanced with two hash tables, FwdTable and RevTable, to create a *stateful pswitch*. The FwdTable and the RevTable record the next hop of a flow indexed by its complete 5-tuple and previous hop. The inP module of

the *pswitch* records the classifier instance selected while processing the first frame of a flow in the FwdTable. While processing a frame that is to be emitted out to a directly attached classifier/server, the outP module of the *pswitch* records the previous hop traversed by the frame as the next hop for frames in the reverse flow direction, in the RevTable. The inP uses the RevTable entry if both FwdTable and rule lookup yield no matches, thus providing a default reverse path containing the same classifier instances as in the forward path. Please see Appendix A.1 for more details.

5.4 Guarantees under Churn

In this section, we argue that the *PLayer* guarantees correct classifier traversal under different kinds of churn – network, policy and classifier churn. Section 5.6 presents a formal analysis of *PLayer* operations and churn guarantees.

5.4.1 Network Churn

The failure or addition of *pswitches* and links constitute network churn. The separation between policy and reachability in the *PLayer* ensures that network churn does not cause policy violations. Every *pswitch* has a copy of the rules encoding the classifiers to be traversed by different traffic, and forwarding of frames is solely done based on these rules. Although frames forwarded to classifiers or servers rendered unreachable by *pswitch* or link failures may be dropped, a classifier will never be bypassed.

Network partitions caused by link or *pswitch* failures concurrent with policy or classifier churn can lead to inconsistencies in the policy and classifier information established at different *pswitches*. We address this problem by employing a 2-stage, versioned policy and classifier information dissemination mechanism, described later in this section.

5.4.2 Policy Churn

Network administrators update policies at a centralized policy controller when the logical topology of the data center network needs to be changed. In this section, we first briefly describe our policy dissemination mechanism. We then discuss possible classifier traversal violations and how we successfully prevent them.

Policy Dissemination

The policy controller reliably disseminates policy information over separate TCP connections to each *pswitch*. If this step fails due to a network partition between the policy controller and some *pswitch*, then the update is canceled and the administrator is notified. After all *pswitches* have received the complete policy information, the policy controller sends a signal that triggers each *pswitch* to adopt the latest update. The signal, which is conveyed in a single packet, has a better chance of synchronously reaching the different *pswitches* than the multiple packets carrying the policy information. Similar to network map dissemination [75], the policy version number recorded inside encapsulated frames is used to further improve synchronization – a *pswitch* that has not yet adopted the latest policy update will immediately adopt it upon receiving a frame stamped with the latest policy version number. This also makes the dissemination process robust to transient network partitions that cause the trigger signal to be lost.

Policy dissemination over separate TCP connections to each *pswitch* scales well if the number of *pswitches* in the data center is small (a few 100s), assuming infrequent policy updates (a few times a week). If the number of *pswitches* is very large, then the distributed reliable broadcast mechanism suggested by RCP [49] is used for policy dissemination – The policy controller sends policy updates over TCP connections to the *pswitches* directly connected to it. These *pswitches* in turn send the policy information over separate TCP connections to each of the *pswitches* directly connected to them, and so on.

Policy Violations

Frames may be forwarded to classifiers in an incorrect order that violates policy during policy churn, even if policy dissemination is perfectly synchronized. In this section, we illustrate potential violations using some example topologies. In the next section, we describe how we use *intermediate classifier types* to prevent these violations.

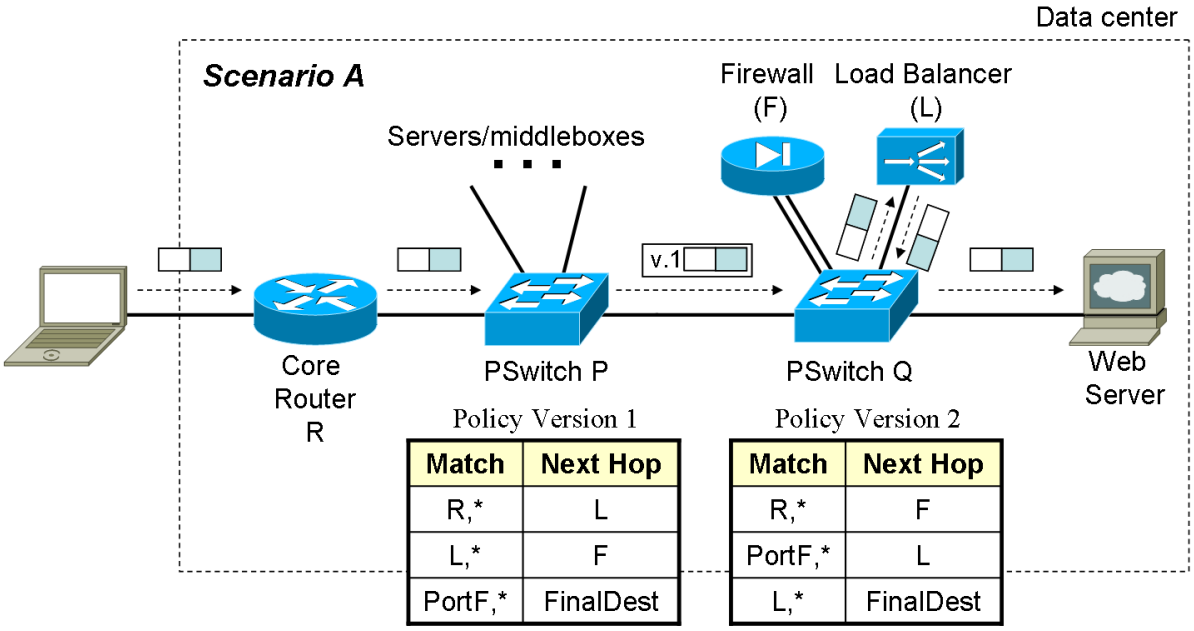

Figure 5.9. Network topology to illustrate policy violations during policy churn. Rule tables correspond to Scenario *A*.

Consider the topology shown in Figure 5.9. Policy version 1 specifies that all traffic entering the data center should first traverse a load balancer followed by a firewall. Policy version 2 reverses the order of classifiers specified in policy version 1, *i.e.,* traffic should first traverse a firewall and then a load balancer. Scenarios *A* and *B*, described below, demonstrate how the lack of perfect time synchronization of policy updates across different *pswitches* causes policy violations. Scenario *C* demonstrates how our support for unmodified classifiers causes policy violations even with perfect time synchronization of policy updates.

- **Scenario A**

  Pswitch P is at policy version 1; *pswitch Q* is at policy version 2, as shown by the

rule tables in Figure 5.9. A frame arriving at *pswitch* P from outside the data center will be forwarded to the load balancer L, as per policy version 1. When *pswitch* Q receives the frame after processing by L, it forwards it to the final destination, as per policy version 2. The frame does not traverse the firewall, thus violating data center security policy. To avoid this violation, *pswitch* Q drops the frame without handing it to L, as the policy version number embedded in it (*i.e.,* 1) is less than Q's current policy version number (*i.e.,* 2).

- **Scenario B**

  Pswitch P is at policy version 2; *pswitch* Q is at policy version 1. A frame arriving at *pswitch* P from outside the data center will be forwarded to the firewall F, as per policy version 2. When *pswitch* Q receives the frame after processing by F, it forwards it to the final destination, as per policy version 1. Although a potentially less crucial classifier L is bypassed in this scenario, the policy violation may still be unacceptable (for example, if L were an intrusion prevention box).

  To avoid the violation, *pswitch* Q updates its current policy (1) to the latest version embedded in the frame (2), before handing it off to F. Now when it receives the frame after processing by F, it correctly forwards it to L, as per policy version 2. If *pswitch* Q had not completely received policy version 2 through the dissemination mechanism before receiving the frame, then it is dropped and not sent to F.

  This mechanism to prevent policy violation will not work if the *red* and *green* interfaces of F are connected to two different *pswitches* Q and T, as shown in Figure 5.10. This is because only *pswitch* Q updates to policy version 2 on seeing the frame with version 2. *Pswitch* T, which receives the frame after processing by F, may remain at policy version 1 and thus incorrectly forwards the frame to the final destination, bypassing L.

- **Scenario C**

  Pswitches P and Q are both at policy version 1. A frame arriving at P from outside the data center is forwarded to L, as per policy version 1. While the frame is being

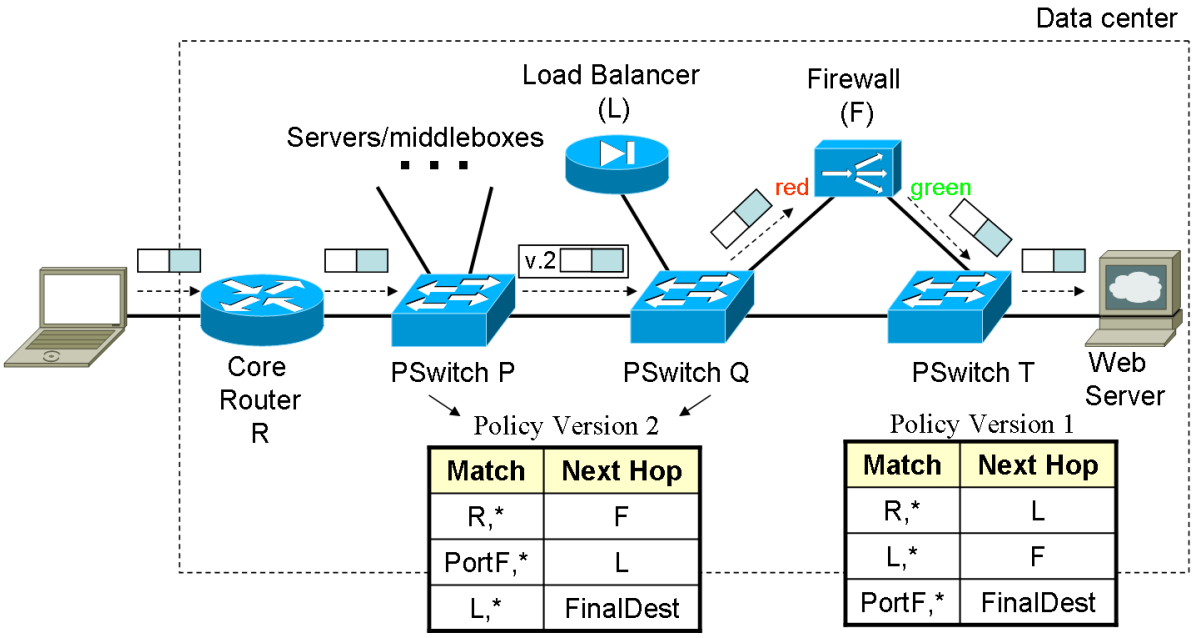

Figure 5.10. Policy violation during churn when the two interfaces of a firewall are connected to different *pswitches*.

processed by L, *pswitches* P and Q both adopt policy version 2 at exactly the same time instant. When the frame arrives at Q after processing by L, it is forwarded to the final destination based on policy 2, bypassing the firewall as in Scenario A. Thus, even perfect synchronization of policy updates will not prevent policy violations.

Irrespective of the policy violations described above, frames will never become stuck in a forwarding loop. Loops in policy specifications are detected and prevented by static analysis during the specification phase itself. The policy version number stamped in frames ensures that each *pswitch* processing a frame uses the latest policy version.

Our mechanisms to prevent policy violations during churn are greatly limited by our support for existing unmodified classifiers. Unmodified classifiers do not preserve the policy version numbers associated with frames they process. If they did (for example, using annotations like in [56]), we can use the policy version number embedded in a frame's *CLayer* to ensure that it is forwarded only based on a single policy during its lifetime. Since classifiers may drop frames or generate new ones, counting the number of frames sent to a classifier

cannot be used to infer the policy version associated with frames output by it. In the next section, we describe how *intermediate classifier types* are used to prevent policy violations.

Intermediate Classifier Types

Specifying conflicting policy updates in terms of *intermediate classifier types* avoids policy violations during policy churn. To avoid the violations discussed in the previous section, we specify the classifier sequence for policy version 2 as *firewall'* followed by *load balancer'*. *firewall'* and *load balancer'* are new classifier types temporarily used during the policy transition period. Although functionally identical to the original classifier types *firewall* and *load balancer*, these *intermediate* classifier types have separate instances, as shown in Figure 5.11. Frames forwarded under policy version 2 traverse these separate classifier instances. Hence, a *pswitch* will never confuse these frames with those forwarded under policy 1, *i.e.,* a frame emitted by L is identified with policy version 1, and a frame emitted by L' is identified with policy version 2. This prevents incorrect forwarding that leads to policy violations. In order to avoid dropping in-flight frames forwarded under policy version 1, rules corresponding to policy version 1, except the one matching new packets entering the data center, are preserved during the policy transition period, as shown in the rule table of Figure 5.11.

Specifying a policy update in terms of intermediate classifier types requires a spare instance of each classifier type affected by the update to be available during the policy transition period. These classifier instances are required only until all frames in flight prior to the policy transition have reached their final destinations, *i.e.,* they are not under processing inside a classifier [4]. After this, the new policy can be re-expressed using the original classifier types *firewall* and *load balancer*. This is equivalent to adding new classifier instances of the intermediate classifier type, a classifier churn scenario that can drop frames. We discuss classifier churn in the next section.

Performing policy updates during off-peak traffic hours reduces the network availability

[4]We assume that a classifier processes a frame in a bounded amount of time.

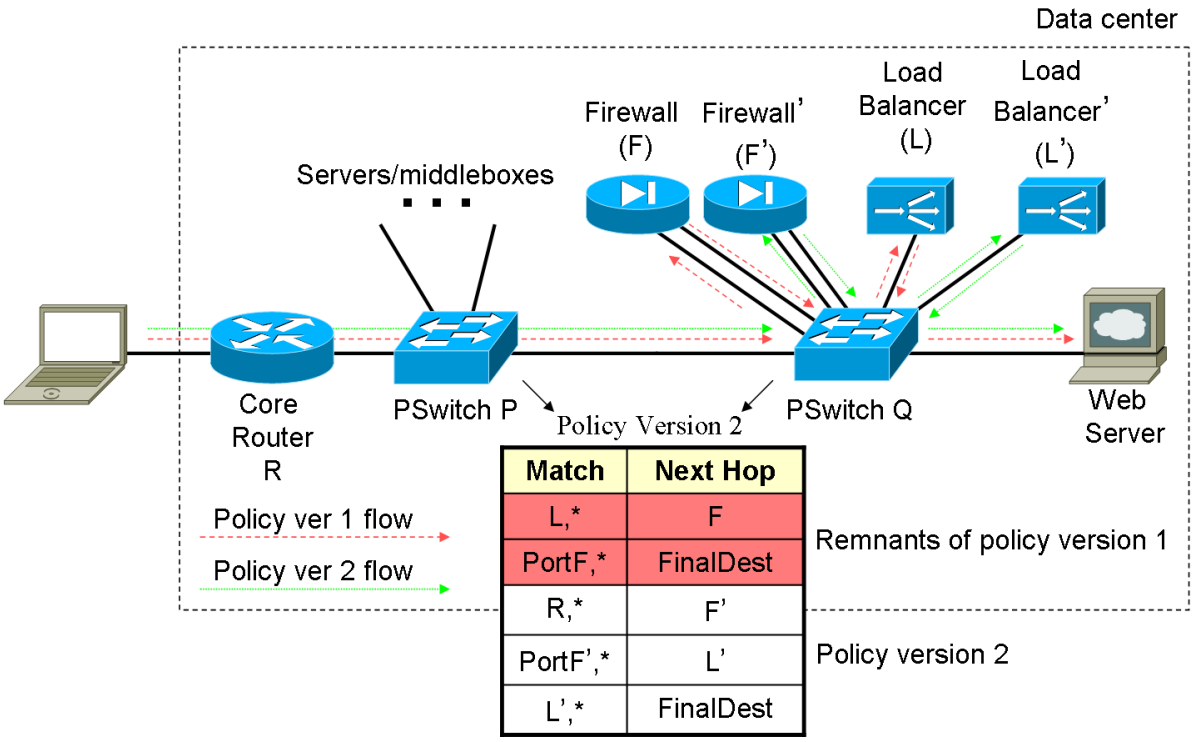

Figure 5.11. Using *intermediate classifier types* to avoid policy violation.

impact of dropped frames. Obtaining instances of intermediate classifier types is also easier. In our example, a second load balancer instance of type *load balancer*, L_2, can be slowly drained of traffic associated with policy version 1, and then reclassified as type *load balancer'*. In order to avoid disturbing flows unrelated to the policy update that traverse the same *load balancer* instances, reclassification is flagged such that it applies only to policies affected by the update.

Using intermediate classifier types only ensures that a particular frame in a flow is not forwarded in a manner that violates policy. It does not ensure that all frames in a flow will be forwarded based on the same policy version. For example, frames entering the data center before *pswitch P*'s transition from policy version 1 to 2 will traverse the classifier sequence specified by policy version 1, while frames arriving after the transition traverse the sequence specified by policy version 2.

If we require all frames in a flow to be forwarded based on the same policy version, we use per-flow state in *pswitches*, as described in Section 5.3.3. Intermediate classifier types are still required when per-flow state is used, in order to prevent policy violations when state expires. However, the policy transition period will be longer when per-flow state is used. This is because the new policy should be re-expressed based on the original classifier types only after all per-flow state created based on the original policy has expired. Otherwise, policy violations are possible when per-flow state is recalculated on state expiration at some intermediate *pswitch* along the flow's path.

Intermediate classifier types and spare classifier instances are not required for non-conflicting updates – *e.g.,* updates that deal with a new traffic type or contain only new classifier types. If classifier traversal inconsistencies during the infrequent and pre-planned policy transition periods are acceptable, then the loose synchronization provided by the policy dissemination mechanisms will alone suffice.

5.4.3 Classifier Churn

Classifier churn, *i.e.,* the failure of existing classifier instances or the addition of new ones, will never cause a policy violation as frames are explicitly forwarded based on policy. However, it affects network availability as some frames may be dropped in certain churn scenarios.

The consistent hashing based classifier instance selection mechanism (Section 5.3.2) ensures that the same classifier instances are selected for all frames in a flow, when no new classifier instances are added. When a running classifier instance fails, all flows served by it are automatically shifted to an active standby, if available, or are shifted to some other instance determined by consistent hashing. If flows are shifted to a classifier instance that does not have state about the flow, it may be dropped, thus affecting availability. However, this is unavoidable even in existing network infrastructures and is not a limitation of the *PLayer*.

Adding a new classifier instance changes the number of instances (n) serving as targets

for consistent hashing. As a result, $\frac{1}{2n}$ of the flows are shifted to the newly added instance, on average. Stateful classifier instances like firewalls may drop the reassigned flow and briefly impede network availability. If n is large (say 5), only a small fraction of flows (10%) are affected. If these relatively small and infrequent pre-planned disruptions are deemed significant for the data center, they can be avoided by enhancing *pswitches* with per-flow state as described in Section 5.3.3.

A stateful *pswitch* uses the next hop entry recorded in the FwdTable for all frames of the flow, thus *pinning* them to the classifier instance selected for the first frame. However, this mechanism will not work in scenarios when a new classifier instance is added at around the same time as one of the following two events: (i) The next hop entry of an active flow is flushed out, (ii) A switch/router failure reroutes packets of the flow to a new *pswitch* which does not have state for the flow.

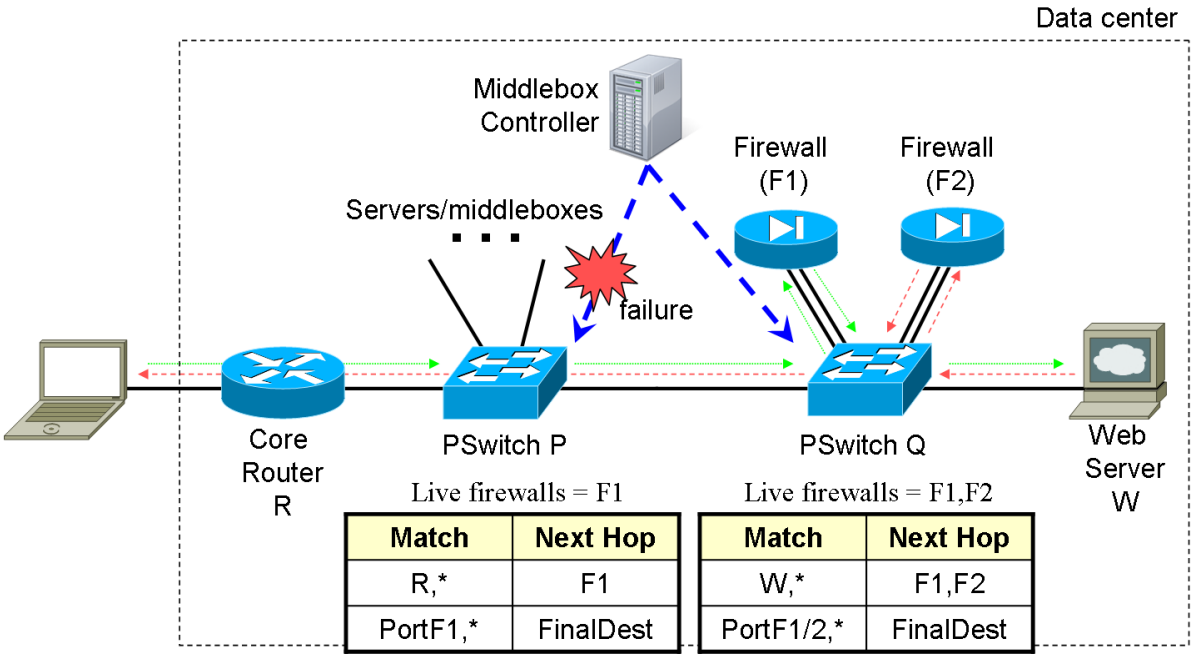

Figure 5.12. Inconsistent classifier information at *pswitches* due to network churn.

Network churn concurrent with classifier churn may lead to differing classifier status information at different *pswitches*, as shown in Figure 5.12. *Pswitch P* did not receive

the information that firewall instance $F2$ has become alive because the classifier controller could not reach P due to a network partition. Thus *pswitch* P selects the firewall instance $F1$ for all frames entering the data center. For the reverse flow direction (web server $\rightarrow$ external client), *pswitch* Q selects the firewall instance $F2$ for approximately half of the flows. $F2$ will drop these frames as it did not process the corresponding frames in the forward flow direction and hence lacks the required state. Although this inconsistency in classifier information does not cause policy violations, we can reduce the number of dropped frames by employing a 2-stage classifier information dissemination mechanism similar to the policy dissemination mechanism described in Section 5.4.2 – Classifier status updates are versioned and a *pswitch* adopts the latest version on receiving a signal from the classifier controller or on receiving a frame with an embedded classifier status version number greater than its current version.

5.5 Implementation and Evaluation

In this section, we briefly describe our prototype implementation of the *PLayer*. We then demonstrate its functionality and flexibility under different network scenarios, as well as provide preliminary performance benchmarks.

5.5.1 Implementation

We have prototyped *pswitches* in software using Click [72]. An unmodified Click *Etherswitch* element formed the Switch Core, while the Policy Core (Click elements) was implemented in 5500 lines of C++. Each interface of the Policy Core plugs into the corresponding interface of the Etherswitch element, maintaining the modular switch design described in Section 5.2.

Due to our inability to procure expensive hardware classifiers for testing, we used the following commercial quality software classifiers running on standard Linux PCs:

1. *Netfilter/iptables* [41] based firewall

2. Bro [83] intrusion detection system

3. *BalanceNG* [4] load balancer

We used the Net-SNMP [26] package for implementing SNMP-based classifier liveness monitoring. Instead of inventing a custom policy language, we leveraged the flexibility of XML to express policies in a simple human-readable format. The classifier controller, policy controller, and web-based configuration GUI were implemented using Ruby-On-Rails [35].

5.5.2 Validation of Functionality

We validated the functionality and flexibility of the *PLayer* using computers on the DETER [47] testbed, connected together as shown in Figure 5.13. The physical topology was constrained by the maximum number of Ethernet interfaces (4) available on individual testbed computers. Using simple policy changes to the *PLayer*, we implemented the different logical network topologies shown in Figure 5.14, without rewiring the physical topology or taking the system offline. Not all devices were used in every logical topology.

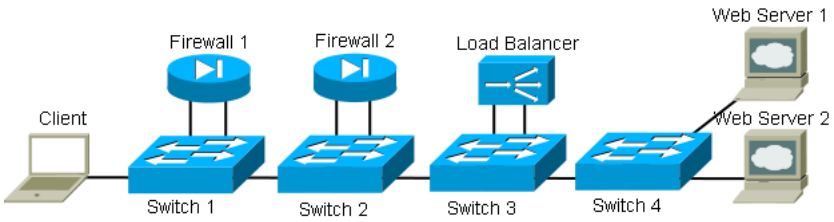

Figure 5.13. Physical topology on the DETER testbed for functionality validation.

Topology A→B:

Logical topology A represents our starting point and the most basic topology – a client directly communicates with a web server. By configuring the policy *[Client, (*,IP_{web1},*,80,tcp)]* → *firewall* at the policy controller, we implemented logical topology B, in which a firewall is inserted in between the client and the web server. We validated that all client-web server traffic flowed through the firewall by monitoring the links. We also observed that all flows were dropped when the firewall failed (was turned off).

Topology B→C:

Adding a second firewall, Firewall 2, in parallel with Firewall 1, in order to split the processing load resulted in logical topology C. Implementing logical topology C required no policy changes. The new firewall instance was simply registered at the classifier controller, which then immediately informed all four *pswitches*. Approximately half of the existing flows shifted from Firewall 1 to Firewall 2 upon its introduction. However, no flows were dropped as the filtering rules at Firewall 2 were configured to temporarily allow the pre-existing flows. Configuring firewall filtering behavior is orthogonal to *PLayer* configuration.

Topology C→B→C:

To validate the correctness of *PLayer* operations when classifiers fail, we took down one of the forwarding interfaces of Firewall 1, thus reverting to logical topology B. The SNMP daemon detected the failure on Firewall 1 in under 3 seconds and immediately reported it to all *pswitches* via the classifier controller. All existing and new flows shifted to Firewall 2 as soon as the failure report was received. After Firewall 1 was brought back alive, the *pswitches* restarted balancing traffic across the two firewall instances in under 3 seconds.

Topology C→D:

We next inserted a load balancer in between the firewalls and web server 1, and added a second web server, yielding logical topology D. Clients send HTTP packets to the load balancer's IP address IP_{LB}, instead of a web server IP address (as required by the load balancer operation mode). The load balancer rewrites the destination IP address to that of one of the web servers, selected in a round-robin fashion. To implement this logical topology, we specified the policy *[Client, (*,IP_{LB},*,80,tcp)]* → *firewall* and the corresponding reverse policy for the client-load balancer segment of the path. The load balancer, which automatically forwards packets to a web server instance, is not explicitly listed in the classifier sequence because it is the end point to which packets are addressed. We also specified the policy *[Web, (*$IP_{web1/2}$*,*,80,*,tcp)]→load balancer*. This policy enabled us to force the web servers' response traffic to pass through the load balancer without reconfiguring the default IP gateway on the web servers, as done in current best practices. We verified that

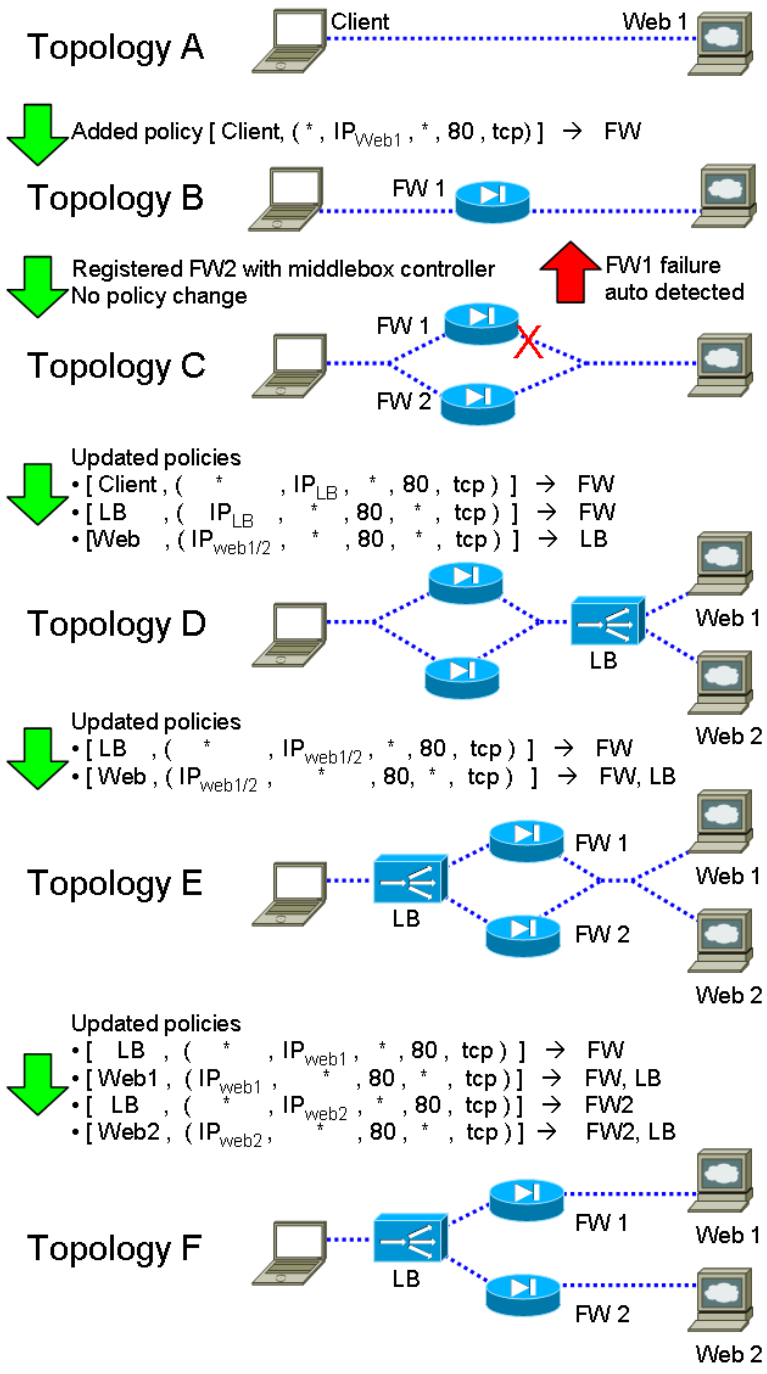

Figure 5.14. Logical topologies used to demonstrate *PLayer* functionality.

the client-web server traffic was balanced across the two firewalls and the two web servers. We also verified the correctness of *PLayer* operations under firewall, load balancer and web server failure.

Topology D→E:

In order to demonstrate the *PLayer* 's flexibility, we flipped the order of the firewalls and the load balancer in logical topology D, yielding topology E. Implementing this change simply involves updating the policies to *[LB, (*,$IP_{web1/2}$,*,80,tcp)]* → *firewall* and *[Web, ($IP_{web1/2}$,*,80,*,tcp)]* → *firewall,* load balancer. We do not specify a policy to include the load balancer on the client to web server path, as the HTTP packets sent by the client are addressed to the load balancer, as before.

Topology E→F:

To further demonstrate the *PLayer*'s flexibility, we updated the policies to implement logical topology F, in which Firewall 1 solely serves web server 1 and Firewall 2 solely serves web server 2. This topology is relevant when the load balancer intelligently redirects different types of content requests (for example, static versus dynamic) to different web servers, thus requiring different types of protection from the firewalls. To implement this topology, we changed the classifier type of Firewall 2 to a new type $firewall_2$, at the classifier controller. We then updated the forward direction policies to *[LB, (*,IP_{web1},*,80,tcp)]* → *firewall* and *[LB, (*,IP_{web2},*,80,tcp)]* → $firewall_2$, and modified the reverse policies accordingly.

Although the experiments described above are limited to simple logical topologies and policies on a small testbed, the logical topology modifications and failure scenarios studied here are orthogonal to the complexity of the system. We further validated the functionality and correctness of the *PLayer* in a larger and more complex network topology similar to the popular data center topology shown in Figure 2.1. Due to the limited number of physical network interfaces on our test computers, we emulated the desired layer-2 topology using UDP tunnels. We created *tap* [93] interfaces on each computer to represent virtual layer-2 interfaces, with their own virtual MAC and IP addresses. The frames sent by an unmodified

application to a virtual IP address reaches the host computer's *tap* interface, from where it is tunneled over UDP to the *pswitch* to which the host is connected in the virtual layer-2 topology. Similarly, frames sent to a computer from its virtual *pswitch* are passed on to unmodified applications through the *tap* interface. *Pswitches* are also inter-connected using UDP tunnels.

For a more formal analysis of *PLayer* functionality and properties, please see Section 5.6.

5.5.3 Benchmarks

In this section, we provide throughput and latency benchmarks for our prototype *pswitch* implementation, relative to standard software Ethernet switches and on-path classifier deployment. Our prototype implementation focuses on feasibility and functionality, rather than optimized performance. While the performance of a software *pswitch* may be improved by code optimization, achieving line speeds is unlikely. The 50x speedup obtained when moving from a software to hardware switch prototype in [51] offers hope that a future hardware-based *pswitch* implementation [27] will achieve line rates. We believe that the hardware *pswitch* implementation will have sufficient switching bandwidth to support frames traversing the switch multiple times due to classifiers and will be able to operate at line speeds.

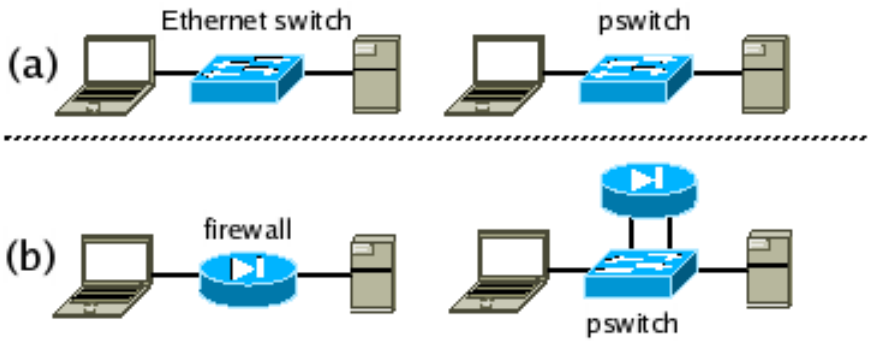

Figure 5.15. Topologies used in benchmarking *pswitch* performance.

Our prototype *pswitch* achieved 82% of the TCP throughput of a regular software Ethernet switch, with a 16% increase in latency. Figure 5.15(a) shows the simple topology used

in this comparison experiment, with each component instantiated on a separate 3GHz Linux PC. We used *nuttcp* [29] and *ping* for measuring TCP throughput and latency, respectively. The *pswitch* and the standalone Click Etherswitch, devoid of any *pswitch* functionality, saturated their PC CPUs at throughputs of 750 Mbps and 912 Mbps, respectively, incurring latencies of 0.3 ms and 0.25 ms.

A classifier deployment using our prototype *pswitch* achieved only 40% of the throughput of a traditional on-path classifier deployment, while doubling the latency. Figure 5.15(b) shows the simple topology used in this comparison experiment. The on-path firewall deployment achieved an end-to-end throughput of 932 Mbps and a latency of 0.3 ms, while the *pswitch*-based firewall deployment achieved 350 Mbps with a latency of 0.6 ms. Although latency doubled as a result of multiple *pswitch* traversals, the sub-millisecond latency increase is in general much smaller than wide-area Internet latencies. The throughput decrease is a result of packets traversing the *pswitch* CPU twice, although they arrived on different *pswitch* ports. Hardware-based *pswitches* with dedicated multi-gigabit switching fabrics should not suffer this throughput drop.

Microbenchmarking showed that a *pswitch* takes between 1300 and 7000 CPU ticks [5] to process a frame, based on its destination. A frame entering a *pswitch* input port from a classifier or server is processed and emitted out of the appropriate *pswitch* output ports in 6997 CPU ticks. Approximately 50% of the time is spent in rule lookup (from a 25 policy database) and classifier instance selection, and 44% on frame encapsulation. Overheads of packet classification and packet handoff between different Click elements consumed the remaining INP processing time. An encapsulated frame reaching the *pswitch* directly attached to its destination server/classifier was decapsulated and emitted out to the server/classifier in 1312 CPU ticks.

[5] We counted the CPU ticks consumed by different *pswitch* operations using the RDTSC x86 instruction on a 3GHz desktop PC running Linux in single processor mode (3000 ticks = 1 microsecond). Due to variability in CPU tick count caused by other processes running on the PC, we report the minimum CPU tick count recorded in our repeated experiment runs as an upper bound on the CPU ticks consumed by *pswitch* operations.

5.6 Formal Analysis

In this section, we validate the functionality of the *PLayer* and discuss its limitations using a formal model of policies and *pswitch* forwarding operations.

5.6.1 Model

Network administrators require different types of traffic to go through different sequences of classifiers. These requirements can be expressed as a set of policies, of the form:

$$\text{Traffic Type } i \quad : \quad M_{i_1}, M_{i_2}, \ldots, M_{i_j}, \ldots, M_{i_{n_i}}, F$$

where M_{i_j} is a classifier type (say, firewall), F denotes the final destination, and n_i is the number of classifiers to be traversed by traffic type i. Note that F is only a place-holder for the final destination; the final destination of a frame is determined by its destination MAC and/or IP addresses.

The *PLayer* uses 5-tuple based classification as a simple and fast mechanism to identify traffic types. Thus, *PLayer* policies are of the form:

$$(S_i, C_i) \quad : \quad M_{i_1}, M_{i_2}, \ldots, M_{i_j}, \ldots, M_{i_{n_i}}, F$$

where C_i is the *Traffic Selector*, a 5-tuple based classifier that identifies traffic type i, and S_i is the *Start Location*, denoting where the frame arrived from (*e.g.,* a border router or an internal server).

5-tuple based traffic type identification is affected by classifiers that modify packet headers. Hence, a *PLayer* policy for traffic type i that includes such classifiers is expressed as a sequence of *per-segment policies*, as shown below:

$$\begin{aligned}
(S_i, C_{i1}) \quad &: \quad M_{i_1}, \ldots, M_{i_{k_1}} \\
(M_{i_{k_1}}, C_{i2}) \quad &: \quad M_{i_{k_1+1}}, M_{i_{k_1+2}}, \ldots, M_{i_{k_2}} \\
&\ldots \\
(M_{i_{k_{s-1}}}, C_{is}) \quad &: \quad M_{i_{k_{s-1}+1}}, M_{i_{k_{s-1}+2}}, \ldots, M_{i_{k_s}} \\
(M_{i_{k_s}}, C_{i(s+1)}) \quad &: \quad M_{i_{k_s+1}}, M_{i_{k_s+2}}, \ldots, M_{i_{n_i}}, F
\end{aligned}$$

where $M_{i_{k_1}} \ldots M_{i_{k_s}}$ are the classifier types that modify packet headers, and $(M_{i_{k_{j-1}}}, C_{ij})$ matches packets modified by classifier type $M_{i_{k_{j-1}}}$.

Each per-segment policy is converted into a series of forwarding rules stored in rule-tables at each *pswitch*. A forwarding rule specifies the next hop for a packet arriving from a particular previous hop that matches a particular classifier. For example, the per-segment policy with classifier C_{i1} above results in the following forwarding rules:

$$\begin{aligned} S_i, C_{i1} &: M_{i_1} \\ M_{i_1}, C_{i1} &: M_{i_2} \\ &\ldots \\ M_{i_{k_1-1}}, C_{i1} &: M_{i_{k_1}} \end{aligned}$$

The *path* taken by a frame f is denoted by

$$\text{path}(f) = e_1, e_2, \ldots, e_i, \ldots, e_l, F/D$$

where entities $e_1 \ldots e_l$ are classifier instances. F/D denotes that the frame reached its final destination (F) or was dropped (D).

Each classifier type M_i has T_i instances $m_{i1}, m_{i2}, \ldots, m_{iT_i}$. For example, $\text{path}(f) = m_{22}, m_{13}, F$ implies that frame f traversed the instance 2 of classifier type M_2, instance 3 of M_1, and then reached its final destination. $\text{path}(f) = m_{22}, D$ implies that it got dropped before reaching its final destination. The drop may have been the result of m_{22}'s functionality (*e.g.,* firewalling) or because of lack of network connectivity or non-availability of active classifier instances. We do not consider frame drops caused by classifier functionality in this analysis.

Pswitches dictate the path taken by a frame. When a *pswitch* receives a frame from a classifier or server, it looks up the forwarding rule that best matches it. Based on the forwarding rule, it forwards it to an instance of the specified classifier type or to the final

destination, or drops it. This operation can be represented as follows:

$$\begin{aligned} \text{path}(f) &\rightarrow \text{path}(f).m_x \\ &\text{or} \\ \text{path}(f) &\rightarrow \text{path}(f).F \\ &\text{or} \\ \text{path}(f) &\rightarrow \text{path}(f).D \end{aligned}$$

where m_x is an instance of the classifier type M_x specified by the matching forwarding rule. '.' represents path concatenation.

The previous hop of a frame is identified based on its source MAC address, the *pswitch* interface it arrived on, or its VLAN tag. Any rule lookup mechanism can be employed, as long as all *pswitches* employ the same one, *i.e.,* two *pswitches* with the identical rule tables will output the same matching rule for a frame.

A *pswitch* selects a classifier instance for a frame by calculating a flow direction agnostic consistent hash on its 5-tuple fields that are unmodified by the classifier. This information is also included with the policies and forwarding rules (not shown above for clarity). This implies that a *pswitch* will select the same classifier instance for all frames in the same flow, in either direction. This also implies that the classifier instance selection is independent of the *pswitch* on which it is done, if all *pswitches* have the same classifier database.

5.6.2 Desired Properties

Correctness

$\text{path}(f) = m_{r_1 s_1}, m_{r_2 s_2}, \ldots, m_{r_l s_l}, F$ is *correct* , if $m_{r_i s_i} \in M_{r_i}$, where $M_{r_1}, M_{r_2}, \ldots, M_{r_l}$ is the classifier sequence associated with frame f's traffic type. $\text{path}(f) = m_{r_1 s_1}, m_{r_2 s_2}, \ldots, m_{r_l s_l}, D$ is *correct* , if $M_{r_1}, M_{r_2}, \ldots, M_{r_l}$ is a (proper or not proper) prefix of the classifier sequence associated with f's traffic type.

Consistency

path(f) is *consistent* if for all frames g in the same flow and same direction as frame f, path(g) = path(f).

Availability

The availability of the *PLayer* is the fraction of frames that reach their final destination, *i.e.,* $\frac{|\{f|F \in \text{path}(f)\}|}{|\{f\}|}$.

Next, we analyze how well the *PLayer* satisfies the above properties under various scenarios, using the model developed so far.

5.6.3 Policy Churn

A new policy j *conflicts* with an existing policy i, if the following conditions are satisfied:

1. $C_{ix} \cap C_{jy} \neq \phi$, and
2. $S_{ix} = S_{jy}$

where (S_{ix}, C_{ix}) and (S_{jy}, C_{jy}) are the previous hops and traffic classifiers associated with the forwarding rules of policies i and j. The intersection of two classifiers is the set of all packets that can match both. For example, the intersection of classifiers srcip = 128.32.0.0/16 and dstport = 80 includes the packet from 128.32.123.45 destined to port 80. As another example, the intersection of classifiers dstport = 80 and dstport = 443 is empty.

Conflicting policies, and thus their forwarding rules, are specified in terms of *intermediate* classifier types $M'_1, M'_2, \ldots$. A *pswitch* will never match an inflight frame f with path(f) = $e_1, e_2, \ldots, e_i$ against a forwarding rule of the new policy, as the previous hop e_i is an instance of one of the original classifier types $M_1, M_2, \ldots$. In other words, we avoid conflicting policies by ensuring that the condition $S_{ix} = S_{jy}$ never holds. If the original policy is no longer present in the *pswitch* rule table, then f is dropped. Instances of the original classifier types are re-used only after allowing sufficient time for all in-flight packets re-directed under the original policy to reach their destinations. Thus, correctness is guar-

anteed, although availability is reduced. Note that this correctness guarantee is independent of the policy dissemination mechanism.

5.6.4 Addition or Failure of Network Links

The addition or failure of network links only affects the ability of a *pswitch* to forward a frame to its next hop, and not the selection of the next hop. Thus, the path of a frame may get truncated, but correctness and consistency are unaffected. The availability lost due to frames not reaching their final destination is attributable to the underlying forwarding mechanisms used by the *PLayer* and not to the *PLayer* itself.

5.6.5 Inaccuracies in Classifier or Previous Hop Identification

Correctness of *PLayer* operations critically depends on accurate traffic classification (*i.e.,* well-defined C_is) and previous hop identification (*i.e.,* accurate detection of S_is). Section 5.7 discusses how the *PLayer* addresses the limitations caused by inaccuracies in these operations.

5.6.6 Classifier Churn

The addition of a new classifier type does not affect forwarding rule lookup, classifier instance selection or packet forwarding. The *PLayer* allows a classifier type to be deleted only after all policies including it are deleted. Hence correctness, consistency and availability are unaffected by the addition or deletion of a classifier type.

The planned removal or failure of an instance of a particular classifier type affects only the classifier instance selection operation of a *pswitch*. Flows whose frame 5-tuples hashed to the removed classifier instance will now be shifted to a different instance of the same classifier type, or dropped if no instance is available. Thus consistency and availability are hampered. However, this is inevitable even in today's mechanisms[6].The addition of a new

[6]Active standby classifier instances can be used, if available. Although *consistency* as defined here is violated, in practice the packets do not get dropped.

classifier instance also impacts the classifier instance selection process only. Some flows will now hash to the new instance and thus get shifted there. This again impacts consistency and availability (because stateful classifiers may drop these packets), but correctness is preserved. We assume that, in the worst case, a classifier receiving an unexpected frame in the middle of a flow simply drops it and does not violate any classifier functionality.

5.6.7 Forwarding Loops

The *PLayer* cannot prevent forwarding loops caused by the underlying forwarding mechanism it uses. However, it does not introduce any forwarding loops of its own. We assume that policies themselves do not dictate forwarding loops. Static analysis of the policy definitions can detect and prevent such policies. A *pswitch* explicitly redirects only those frames received from a classifier or a server. The path(f) of a frame increases and progresses towards its final destination, each time a *pswitch* redirects it. It will never get stuck in a forwarding loop between two *pswitches*. We assume that *pswitches* can accurately identify its interfaces that are connected to other *pswitches*. Since such identification is crucial to existing forwarding mechanisms (like spanning tree construction), automated and manual methods [55] already exist for this purpose.

5.7 Limitations

The following are the main limitations of the *PLayer*:

1. **Indirect Paths**

   Similar to some existing VLAN-based classifier deployment mechanisms, redirecting frames to off-path classifiers causes them to follow paths that are less efficient than direct paths formed by classifiers physically placed in sequence. We believe that the bandwidth overhead and slight latency increase are insignificant in a bandwidth-rich low latency data center network.

2. **Policy Specification**

   Traffic classification and policy specification using frame 5-tuples is not trivial. However, it is simpler than the current ad-hoc classifier deployment best practices. Network administrators specify policies using a configuration GUI at the centralized policy controller. Static policy analysis flags policy inconsistencies and misconfiguration (*e.g.,* policy loops), and policy holes (*e.g.,* absence of policy for SSH traffic). Since every *pswitch* has the same policy set, policy specification is also less complex than configuring a distributed firewall system.

3. **Incorrect Packet Classification**

   5-tuples alone may be insufficient to distinguish different types of traffic if it is obfuscated or uses unexpected transport ports. For example, a *pswitch* cannot identify HTTPS traffic unexpectedly sent to port 80 instead of 443, and forward it to an SSL offload box. Since such unexpected traffic is likely to be dropped by the destinations themselves, classification inaccuracy is not a show-stopper. However, it implies that if deep packet inspection capable firewalls are available, then policies must be defined to forward all traffic to them, rather than allowing traffic to skip firewalls based on their 5-tuples.

4. **Incorrectly Wired Classifiers**

   The *PLayer* requires classifiers to be correctly wired for accurate previous hop identification and next hop forwarding. For example, if a firewall is plugged into *pswitch* interface 5 while the *pswitch* thinks that an intrusion prevention box is plugged in there, then frames emitted to the intrusion prevention box will reach the firewall. Even existing classifier deployment mechanisms critically rely on classifiers being correctly wired. Since classifiers are few in number compared to servers, we expect them to be carefully wired.

5. **Unsupported Policies**

   The *PLayer* does not support policies that require traffic to traverse the same type of classifier multiple times (*e.g., [Core Router, (*,*,*,80,tcp)]* $\rightarrow$ *firewall, load balancer,*

firewall). The previous hop determination mechanism used by *pswitches* cannot distinguish the two firewalls. We believe that such policies are rare, and hence tradeoff complete policy expressivity for simplicity of design. Note that policies involving different firewall types (*e.g., [Core Router, (*,*,*,80,tcp)] → external firewall, load-balancer, internal firewall*) are supported.

6. **More Complex Switches**

   While we believe that the economical implementation of *pswitches* is easily possible given the current state of the art in network equipment, *pswitches* are more complex than regular Ethernet switches.

5.8 Related Work

Indirection is a well-known principle in computer networking. The Internet Indirection Infrastructure [88] and the Delegation Oriented Architecture [94] provide layer-3 and above mechanisms that enable packets to be explicitly redirected through middleboxes located anywhere on the Internet. Due to pre-dominantly layer-2 topologies within data centers, the policy-aware switching layer is optimized to use indirection at layer-2. SelNet [59] is a general-purpose network architecture that provides indirection support at layer '2.5'. In SelNet, endhosts implement a multi-hop address resolution protocol that establishes per flow next-hop forwarding state at classifiers. The endhost and classifier modifications required make SelNet impractical for current data centers. Using per-flow multi-hop address resolution to determine the classifiers to be imposed is slow and inefficient, especially in a data center environment where policies are apriori known. The *PLayer* does not require endhost or classifier modifications. A *pswitch* can quickly determine the classifiers to be traversed by the packets in a flow without performing multi-hop address resolution.

Separating policy from reachability and centralized management of networks are goals our work shares with many existing proposals like 4D [60] and Ethane [51]. 4D concentrates on general network management and does not provide mechanisms to impose off-path clas-

sifiers or to guarantee classifier traversal. Instantiations of 4D like the Routing Control Platform (RCP) [49] focus on reducing the complexity of iBGP inside an AS and not on Data Centers. 4D specifies that forwarding tables should be calculated centrally and sent to switches. The policy-aware switching layer does not mandate centralized computation of the forwarding table – it works with existing network path selection protocols running at switches and routers, whether centralized or distributed.

Predicate routing [85] declaratively specifies network state as a set of boolean expressions dictating the packets that can appear on various links connecting together end nodes and routers. Although this approach can be used to impose classifiers, it implicitly buries classifier traversal policies in a set of boolean expressions, as well as requires major changes to existing forwarding mechanisms.

Ethane [51] is a proposal for centralized management and security of enterprise networks. An Ethane switch forwards the first packet of a flow to a centralized domain controller. This controller calculates the path to be taken by the flow, installs per-flow forwarding state at the Ethane switches on the calculated path and then responds with an encrypted source route that is enforced at each switch. Although not a focus for Ethane, off-path classifiers can be imposed by including them in the source routes. In the *PLayer*, each *pswitch* individually determines the next hop of a packet without contacting a centralized controller, and immediately forwards packets without waiting for flow state to be installed at *pswitches* on the packet path. Ethane has been shown to scale well for large enterprise networks (20000 hosts and 10000 new flows/second). However, even if client authentication and encryption are disabled, centrally handing out and installing source routes in multiple switches at the start of each flow may not scale to large data centers with hundreds of switches, serving 100s of thousands of simultaneous flows[7]. The distributed approach taken by the *PLayer* makes it better suited for scaling to a large number of flows. For short flows (like single packet heartbeat messages or 2-packet DNS query/response pairs), Ethane's signaling and flow setup overhead can be longer than the flow itself. The prevalence of

[7]We estimate that Google receives over 400k search queries per second, assuming 80% of search traffic is concentrated in 50 peak hours a week [43]. Multiple flows from each search query and from other services like GMail are likely to result in each Google data center serving 100s of thousands of new flows/second.

short flows [99] and single packet DoS attacks hinder the scalability of the flow tables in Ethane switches. Although Ethane's centralized controller can be replicated for fault-tolerance, it constitutes one more component on the critical path of all new flows, thereby increasing complexity and chances of failure. The *PLayer* operates unhindered under the current policies even if the policy controller fails.

Some high-end switches like the Cisco Catalyst 6500 [9] allow various classifiers to be plugged into the switch chassis. Through appropriate VLAN configurations on switches and IP gateway settings on end servers, these switches offer limited and indirect control over the classifier sequence traversed by traffic. Classifier traversal in the *PLayer* is explicitly controlled by policies configured at a central location, rather than implicitly dictated by complex configuration settings spread across different switches and end servers. Crucial classifiers like firewalls plugged into a high-end switch may be bypassed if traffic is routed around it during failures. Unlike the *PLayer*, only specially designed classifiers can be plugged into the switch chassis. Concentrating all classifiers in a single (or redundant) switch chassis creates a central point of failure. Increasing the number of classifiers once all chassis slots are filled up is difficult.

MPLS traffic engineering capabilities can be overloaded to force packets through network paths with classifiers. This approach not only suffers from the drawbacks of on-path classifier placement discussed earlier, but also requires classifiers to be modified to relay MPLS labels.

Policy Based Routing (PBR) [34], a feature present in some routers, enables packets matching pre-specified policies to be assigned different QoS treatment or to be forwarded out through specified interfaces. Although PBR provides no direct mechanism to impose classifiers, it can be used along with standard BGP/IGP routing and tunneling to impose classifiers. dFence [76], a DoS mitigation system which on-demand imposes DoS mitigation classifiers on the data path to servers under DOS attack, uses this approach. The *PLayer* does not rely on configurations spread across different routing and tunneling mechanisms. It instead provides a simple and direct layer-2 mechanism to impose classifiers on the data path. A layer-2 mechanism is more suitable for imposing classifiers in a data center, as data

centers are pre-dominantly layer-2 and many classifiers cannot even be addressed at the IP layer.

5.9 Discussion

5.9.1 Leveraging the *CLayer*

The *PLayer* did not utilize the *CLayer* headers as we wanted to support unmodified classifiers for easy deployment in existing data centers. If we have the freedom to modify classifiers, *pswitches* can leverage a packet's *CLayer* header to avoid multiple policy lookups. The first *pswitch* that receives a packet looks up the policy and embeds the sequence of classifiers that the packet must traverse and a pointer to the next hop in its *CLayer* header. Each classifier that processes the packet advances the pointer by one. A *pswitch* that receives the packet can now simply forward the packet to its next hop without re-performing policy lookup and deciphering the previous hop. Since the classifier sequence to be traversed by a frame can now be permanently embedded in it, guaranteeing correct classifier traversal under policy churn becomes trivial.

The modifications required in classifiers to support such propagation of *CLayer* headers are very similar to that required by tracing frameworks like X-trace [56]. The authors in X-trace have integrated propagation of an opaque identifier (the X-trace id in this case) into the TCP/IP processing stack as well as popular libraries like libasync. We hope to leverage their work as part of future work on a clean-slate data center network implementation with classifier support.

5.9.2 Stateless versus Stateful *pswitches*

A stateful *pswitch* offers the following two advantages over a stateless *pswitch*:

1. **Faster next hop determination.**

   A stateful *pswitch* can determine the next hop of a frame faster than a stateless

pswitch. This is because a stateful *pswitch* performs an exact match hash lookup on the FwdTable as against the pattern based rule lookup and subsequent classifier instance selection performed by a stateless *pswitch*, on receiving each packet.

2. **Higher classifier instance selection consistency**

   A stateful *pswitch* provides more consistent classifier instance selection than stateless *pswitches* when new instances of an existing classifier type are added to the network. In a stateless *pswitch*, a classifier instance is selected using consistent hashing on a frame's 5-tuple. As described in Section 5.4.3, when a new classifier instance is added, a small fraction of existing flows may be shifted to the new instance. Stateful classifier instances like firewalls may drop reassigned flows and thus briefly impede network availability. A stateful *pswitch* avoids this flow reassignment by using next hop information recorded in the FwdTable to pin a flow to a particular classifier instance. However, this does not work under the race conditions described in Section 5.4.3.

The main disadvantages of a stateful *pswitch* are its large fast memory requirements and the associated state management complexity. We conservatively estimate that 140MB of fast memory is needed for 1 million flows traversing at most two classifiers in either direction, assuming each next hop entry to consume 24 bytes (13 bytes for 5-tuple + 4 bytes to identify previous hop + 4 bytes to identify next hop + 1 byte for TTL + rest for hash overhead).

5.10 Summary

In this chapter, we designed and prototyped the *PLayer*, an instantiation of policy-based classifier deployment tailored for data centers and enterprise networks. The *PLayer* explicitly redirects traffic to unmodified off-path classifiers specified by policy. The very low latency, high bandwidth data center network environment makes such indirection feasible. Unlike current best practices, our approach guarantees classifier traversal under all network conditions and enables more efficient and flexible data center network topologies. We

demonstrated the functionality and feasibility of the *PLayer* through a software prototype deployed on a small testbed.

Chapter 6

Generic Offload

Classification offload overcomes the scalability limitations of in-place classification. In Chapter 2, we saw that current classification offload solutions are ad-hoc, inflexible, inefficient and hard to configure. To overcome these limitations, we proposed a new *classification layer* (*CLayer*) and an associated *generic offload protocol* in Chapter 4. In this chapter, we describe *GOff*, a specific instantiation of generic classification offload, in detail.

Generic classification offload enables *classifiers* – entities like routers, firewalls and load balancers that traditionally perform packet classification – to shift a portion of classification's computation and memory requirements to *helpers* – endhosts, edge routers, application proxies and other network entities that can aid classification. *Helpers* advertise their capabilities and *classifiers* notify *helpers* about the classification support desired from them. A *helper* on a packet's path embeds the results of classifying the packet in its *CLayer* header. A downstream *classifier* easily reads and uses these results in speeding up its classification operations. For instance, a load balancer implements HTTP session stickiness by requesting an endhost to label all packets in a session with the identity of the web server instance selected for the session. The load balancer easily determines the web server instance for a packet by reading its label. It avoids the overheads of complex layer-7 processing. Generic classification offload thus improves efficiency.

GOff defines a detailed protocol that implements the high-level operations of generic classification offload described above. It has been carefully engineered to make it robust to non-symmetric network paths, path changes, and state discrepancies at participating entities. In addition, it provides mechanisms to detect and deter cheating or malfunctioning *helpers*. *GOff* exposes a BSD sockets-like [87] API that makes it easy to *GOff*-enable existing applications.

We have prototyped *GOff* in software using the Click [72] modular router. We have *GOff*-enabled a variety of existing applications with just few minor modifications to their source code. Our experiments show that *GOff* improves application scalability and performance.

The rest of this chapter is organized as follows. In the next section, we present an overview of *GOff* using multiple examples. Section 6.2 provides additional details about *GOff*, and discusses its feasibility and deployability. In Section 6.3, we describe our prototype *GOff* implementation, and evaluate it both quantitatively and qualitatively in Section 6.4. We present related work in Section 6.5 and summarize this chapter in Section 6.6.

6.1 Overview

There are two types of entities in *GOff*: *classifiers* and *helpers*. *Classifiers* classify a packet and take action depending on the results of the classification. Examples of classifiers are load balancers, firewalls, and routers. *Helpers* aid *classifiers* by performing classification tasks on their behalf. Examples of *helpers* are endhosts that provide HTTP cookies to web load balancers to aid them in session identification, and edge routers in a Diffserv domain which set code points in packet headers to aid core routers in QoS processing. Note that in the latter example, an edge router is both a *helper* and a *classifier*, as it also classifies packets to determine their next hop in addition to helping core routers.

GOff consists of two main components: (i) a signaling protocol and (ii) a new socket API. The signaling protocol is used to coordinate *classifiers* and their *helpers*. It uses the *CLayer* header in a packet to carry signaling information, and the classification "results"

from *helpers* to *classifiers*. A classification result is an opaque bag of bits that is interpreted by the *classifier* to which it is addressed. For instance, to a load balancer, the result can be a label denoting a web server instance. To a firewall, it can be a flag indicating the accept/drop decision for the packet. Application software at endhosts and other *helpers* interact with *GOff* via the new socket API.

6.1.1 Basic Operations with a Single Classifier

We illustrate the basic functionality of *GOff* using the simple scenario shown in Figure 6.1. Endhost A wishes to communicate with endhost B. The router E on the data path between A and B classifies packets based on its QoS policy and assigns them different forwarding priorities. E is thus the *classifier*. It uses the *GOff* signaling protocol to configure *helpers* A and B. Here, *GOff* provides benefits similar to Diffserv [2] and CSFQ [90] – router E does not have to perform expensive packet classification on every packet, nor does it have to maintain per-flow state.

GOff signaling involves a 4-way handshake – *CL_SYN* - *CL_SYNACK* - *CL_ACK1* - *CL_ACK2* – appropriately named to highlight the similarity with TCP's three-way SYN-SYNACK-ACK handshake. When A initiates communication with B, it first sends a *CL_SYN* to B. Through this *CL_SYN*, *helpers* on the $A \rightarrow B$ data-path advertise their capabilities and *classifiers* place classification requests (*ClassReqs*). The *ClassReqs* are echoed back to A in the *CL_SYNACK* generated by B. *Helpers* are notified of the *ClassReqs* addressed to them through the *CL_ACK1* subsequently sent by A. The *helpers* embed the requested classification results in the *CL_ACK1* and subsequent $A \rightarrow B$ data packets. Similarly, *CL_SYNACK* and *CL_ACK2* configure the *helpers* in the $B \rightarrow A$ direction.

Figure 6.1 illustrates the *GOff* signaling messages exchanged between A and B. They are described in detail below:

- **Step 1:**

  A sends a *CL_SYN* to B, advertising its ability to identify packets in the same TCP flow and label them.

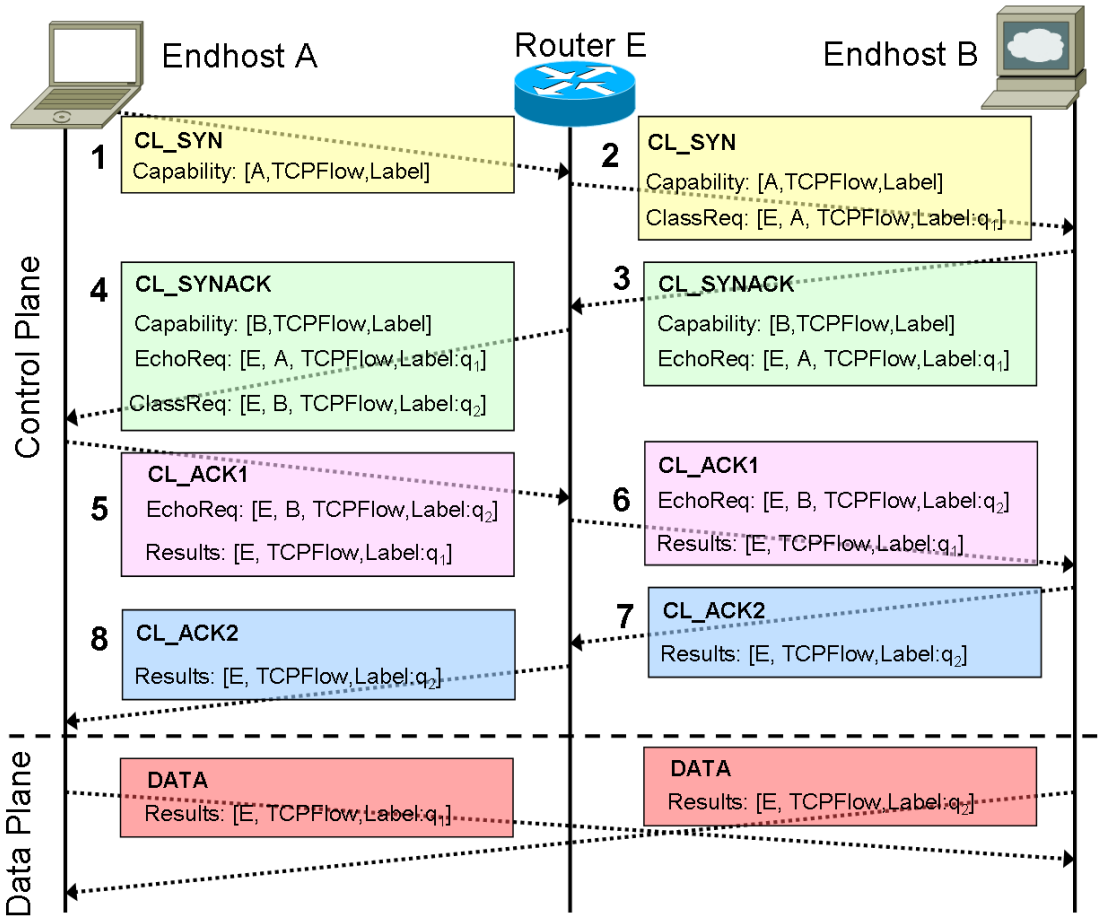

Figure 6.1. Example 1: Detailed *GOff* signaling in a QoS application.

- **Step 2:**

  E forwards the *CL_SYN* after appending a *ClassReq* of the form [*classifier*, *helper*, *classification type*, *action*]. Here, E is requesting A to label all packets in the same *TCPFlow* with the label q_1 that denotes the assigned QoS class.

- **Step 3:**

  B responds to the *CL_SYN* with a *CL_SYNACK*, that advertises its own classification capabilities and echoes the *ClassReq* from the *CL_SYN*.

- **Step 4:**

  E forwards the *CL_SYNACK* after appending a *ClassReq* for labeling packets with q_2, the QoS class for the $B \rightarrow A$ *TCPFlow*.

- **Step 5:**

  A records the *CL_SYNACK EchoReqs* addressed to it with the current TCP flow. It then sends a *CL_ACK1* to *B*, which includes the requested classification result(*i.e., Label*:q_1) and the *ClassReq* copied from the *CL_SYNACK*.

- **Step 6:**

  E forwards the *CL_ACK1* with forwarding priority indicated by the embedded label q_1.

- **Step 7:**

  Like *A*, *B* records the *CL_ACK1 EchoReq* with the current TCP flow, and responds with a *CL_ACK2* that includes the requested classification result *Label*:q_2.

- **Step 8:**

  E simply forwards *CL_ACK2* with forwarding priority q_2, as in Step 6.

Signaling is complete once the *CL_ACK2* reaches *A*. *A* and *B* embed the classification results of their respective *ClassReqs* in every subsequent data packet they exchange.

In this simple scenario, the *CL_ACK2* is unnecessary; a three-way handshake suffices. However, as we describe in detail in Section 6.1.3, *GOff* signaling uses a 4-way handshake to handle non-endhost *helpers* in the presence of network path asymmetry.

The signaling in this example was described as a standalone protocol for simplicity of explanation. In practice, it is piggybacked in the *CLayer* headers of the TCP handshake and initial data packets of a connection, thus avoiding extra round trip overheads. The *CLayer*'s fixed and easily accessible location enables *classifiers* to quickly access the classification results without deep packet inspection, and execute the desired network functionality.

6.1.2 *GOff* API

Applications at endhosts interact with *GOff* using the *GOff* network library. We propose a socket library very similar to the standard BSD socket library [87] so that existing

applications can be easily *GOff*-enabled. However, *GOff* is not restricted to this particular API. A new Internet architecture can use its own API [88, 53].

Our library consists of functions like **clconnect** and **clbind** that have direct semantic correspondence with the standard BSD socket library, and *GOff*-specific functions like **clsessioncreate**. A *GOff*-aware application has a structure very similar to that of any common application using the BSD library. Listing 1 shows the outline of a simple TCP listener and sender program that use the *GOff* socket library.

Algorithm 1 Using the *GOff* socket library

```
 1: procedure LISTENER
 2:     claddcapability(TCP_FLOW, LABEL)
 3:     s = clsocket(AF_INET, SOCK_STREAM,0)
 4:     clbind(s, &my_addr, sizeof(struct sockaddr))
 5:     listen(s, 10)
 6:     new_fd = claccept(s, (struct sockaddr *)&their_addr, &their_addr_size, &new_sess_handle)
 7:     recv(new_fd, recv_buf, recv_buflen,0)
 8:     send(new_fd, send_buf, send_buf_len, 0)
 9:     clclose(new_fd)
10: end procedure
11:
12: procedure SENDER
13:     claddcapability(TCP_FLOW, LABEL)
14:     s = clsocket(AF_INET, SOCK_STREAM, 0)
15:     clsessioncreate(TCP_FLOW, &sess_handle)
16:     clconnect(s, their_addr, sizeof(struct sockaddr), sess_type, sess_handle)
17:     recv(s, recv_buf, recv_buflen, 0)
18:     send(s, send_buf, send_buflen, 0)
19:     clclose(s)
20: end procedure
```

claddcapability enables an application to advertise its semantic classification capabilities (*e.g.,* 'I can identify the HTTP session of a packet'). **clsessioncreate** creates new session state of the specified type (*e.g.,* TCP_FLOW) and returns a handle to the application. The application associates a TCP connection with a session, by passing the session

handle to **clconnect**(). For the first TCP connection in a session, the *GOff* signaling protocol is piggybacked on the standard TCP SYN-SYNACK-ACK exchange. The application sends and receives data using standard *send* and *recv* socket calls. The *GOff* processing module in the OS performs the classification tasks configured during *GOff* signaling on the packets generated by *send()* before emitting them out. This module also strips out *CLayer* headers from received packets and updates session state before handing them to the OS network stack.

6.1.3 Multiple Classifiers and Leveraging *Helper* Context

We now describe how *GOff* provides a generic mechanism that simultaneously supports network operations requiring different kinds of classification – IP forwarding, QoS, and load balancing. This example also demonstrates how application semantic context available at a *helper* can simplify the processing load at a *classifier*, specifically a web load balancer.

A web load balancer implements HTTP session *stickiness*, *i.e.,* it forwards all TCP connections in the same HTTP session to the same web server instance. This is necessary for application correctness when local state (shopping cart contents, SSL session keys) is involved, and provides performance gains due to caching when state is centralized. Current solutions rely on HTTP cookies [15]. On receiving an HTTP connection that does not include a cookie identifying a web server instance, a load balancer carefully injects a 'set cookie request' into the corresponding HTTP response. This cookie contains the id of the web server instance selected by the load balancer. End hosts include the cookie in subsequent HTTP connections. The load balancer reads the cookie and selects the same web server instance.

Current cookie-based solutions leverage semantic context at end hosts for HTTP session identification. However, they are complex and inefficient. Injecting cookies into an HTTP connection's TCP stream is hard. Reading the cookie value requires complex layer-7 processing. *GOff* seamlessly leverages session context available at endhosts to provide the same

session stickiness without requiring the complex TCP stream reconstruction and application header parsing required by current solutions.

Figure 6.2 illustrates the scenario we describe next. A web browser (say, *Firefox*) at endhost A sends HTTP requests to *httpd* running on a webserver in the server farm W. Load balancer L spreads out HTTP requests from endhosts across the web servers in the farm. As in HTTP cookies, L leverages the semantic context available at A to send all its HTTP requests to the same webserver W_1. Edge router E performs prioritized traffic forwarding with help from A and W_1, as in Section 6.1.1. To reduce processing and memory requirements, core router C offloads IP route lookup to less-loaded edge routers E and F, as in MPLS. In summary, E, C and L are *classifiers* and A, W_1, E and F are *helpers*.

Before opening the first HTTP connection to L's public IP address (IP_L), Firefox at A creates a new *GOff* session. It includes the session handle in all TCP connections associated with same HTTP session. We now explain how the various *classifiers* and *helpers* participate in the *GOff* signaling exchange piggybacked on the first TCP connection.

CL_SYN

Figure 6.2**(1)** shows the *CL_SYN* that reaches W_1, after the processing described below:

1. A sends out a *CL_SYN* advertising its *HTTPSess* and *TCPFlow* classification capabilities.

2. In addition to inserting a *ClassReq* as in Section 6.1.1, E also advertises its ability to group packets based on their destination IP in the *CL_SYN*, before forwarding it on.

3. C looks up the route for the *CL_SYN* and appends a *ClassReq* directing E to label all packets destined to IP_L with a label x.

4. F also advertises its destination based packet grouping ability like E, but does not insert a *ClassReq* as it is not QoS-aware.

5. L selects a webserver W_1 based on current load conditions, and forwards the *CL_SYN*

to it after appending a *ClassReq* for A to label all packets in the same HTTP session with W_1.

CL_SYNACK

Figure 6.2**(2)** shows the *CL_SYNACK* received by A, after the processing described below:

1. W_1 responds with a *CL_SYNACK* advertising its capability list and echoing the three *ClassReqs* from the *CL_SYN*.
2. L forwards on the *CL_SYNACK* unmodified, as it does not classify packets destined to endhosts.
3. F appends its capability advertisement.
4. C appends a *ClassReq* for F to label all packets to IP_A with y.
5. E appends a *ClassReq* to label all packets in the same TCP flow with q_2.

CL_ACK1

Figure 6.2**(3)** shows the *CL_ACK1* received by W_1, after the processing described below:

1. A records the *EchoReqs* in the received *CL_SYNACK* that are addressed to it. It copies the other *EchoReqs* to the *InstallReqs* of the *CL_ACK1* it next sends to IP_L. The *CL_ACK1* also contains the requested classification results (*Label*:q_1 and *Label*:W_1) and the *EchoReqs* from the *CL_SYNACK*.
2. E records the *InstallReq* addressed to it and appends the specified classification result (*Label*:x).
3. C uses the label x to quickly identify the outgoing interface for the *CL_ACK1*, without re-performing complex route lookup or traffic engineering.

4. F forwards the *CL_ACK1* unmodified.

5. L quickly selects the webserver instance W_1 based on the label and forwards the packet there. No TCP stream reconstruction or HTTP parsing is necessary.

CL_ACK2

Figure 6.2**(4)** shows the *CL_ACK2* received by A, after the processing described below:

1. W_1 records the *EchoReqs* addressed to it by E in the *CL_ACK1*. The remaining *EchoReq*, addressed to F, is copied into the *InstallReqs* of the *CL_ACK2* generated by W_1. The *CL_ACK2* also includes the classification result *Label*:q_2.

2. L relays the *CL_ACK2* unmodified.

3. F records the *InstallReq* addressed to it and includes the requested classification result *Label*:y.

4. C uses this label to quickly forward the packet.

5. E uses the label q_2 to forward the packet to A with appropriate QoS. This completes the *CLayer* signaling protocol.

The existence of asymmetric network paths (for example, due to Internet path diversity [64] or load balancing Direct Server Return mode [74]) creates the need for the *CL_ACK2* message and makes *GOff* signaling 4-way instead of 3-way. A non-endhost *helper* reads the *ClassReqs* addressed to it in the $A \rightarrow B$ direction from the *InstallReqs* in a *CL_ACK1*, and not from the *EchoReqs* field of a *CL_SYNACK*, as the *CL_SYNACK* may take a different network path that omits the *helper*. Thus, we need the fourth signaling message – *CL_ACK2* – to inform *helpers* about $B \rightarrow A$ *ClassReqs*.

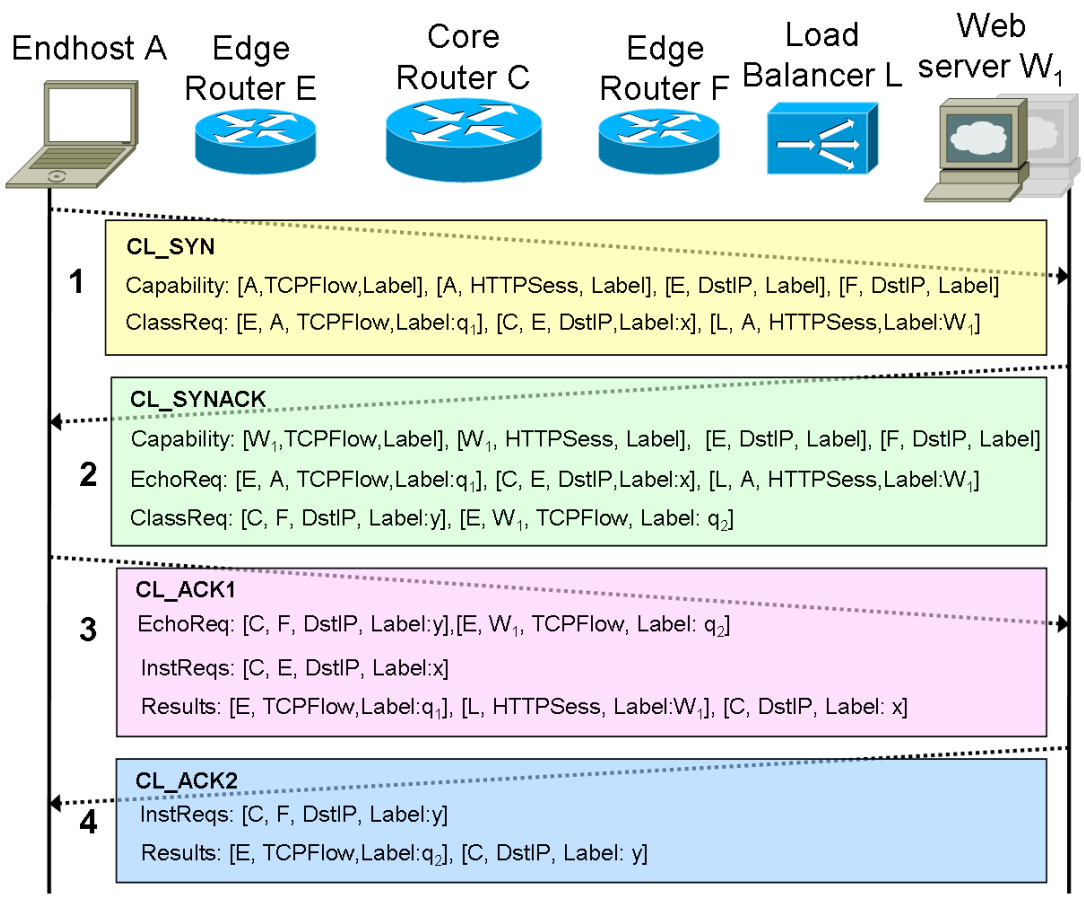

Figure 6.2. Example 2: Using *GOff* to offload QoS labeling, IP route lookup and HTTP session identification to endhosts and edge routers.

6.1.4 Pro-actively helping *Classifiers*

Instead of waiting for a *classifier* (like the load balancer in the previous section) to request classification support, a *helper* may pro-actively include classification hints in the *CLayer* of packets it originates. For example, an endhost originating a packet to a destination known to use anycast proactively includes hints about its geographic location (say, GPS coordinates) in its *CLayer*. These hints aid anycast routers to forward the packet to the closest service replica. The routers use regular *GOff* signaling to instruct the endhost to insert a shorter label identifying the chosen replica in subsequent packets it sends.

As another example, the web browser at an endhost may proactively include a label identifying the type of content requested in its HTTP request – image, video, HTML or

just the extension of the requested file. A content-aware load balancer can quickly forward a request containing such a label to the web server specialized for the requested content type without reconstructing the TCP stream and parsing HTTP headers to identify the content type.

When communicating with a particular destination for the first time, a web browser is unaware of the load balancer's labeling scheme. It thus sends unlabeled packets. The load balancer processes these packets on its slow path. It conveys the labeling schema to the web browser using *GOff*. The web browser caches this information similar to how it stores HTTP cookies, and uses it to label subsequent connections.

This approach to pre-classify content types is limited to a small standard set of classification schema. Moreover, the rules and their semantics cannot be instantaneously changed at the load balancers since they are cached by endhosts.

6.1.5 Explicit Coordination between *helpers*

Using HTTP cookies to identify HTTP sessions in the previous example does not incur too much additional overhead if L anyway parses HTTP headers to perform URL-based content filtering. We now revisit the scenario from Section 2.2 where the extra processing and state demanded by classification imposes a high overhead over normal operations. This example also demonstrates how on-path *classifiers* can explicitly coordinate and signal each other in order to establish common state at different *helpers*.

Figure 6.3 illustrates the scenario we describe next. Endhost A wishes to communicate with FTP server B located behind a firewall farm. Load balancers L and M distribute traffic across the different firewalls. For correct firewall functionality, packets in forward and reverse flow directions, as well as in both control and data flows of an FTP session, must be processed by the same firewall. In current mechanisms [74], M records the link on which a packet arrived and uses the recorded information to choose the outgoing link for a packet in the reverse direction. In addition, L and M must be capable of reconstructing TCP streams and parsing FTP headers in order to identify the control and data connections

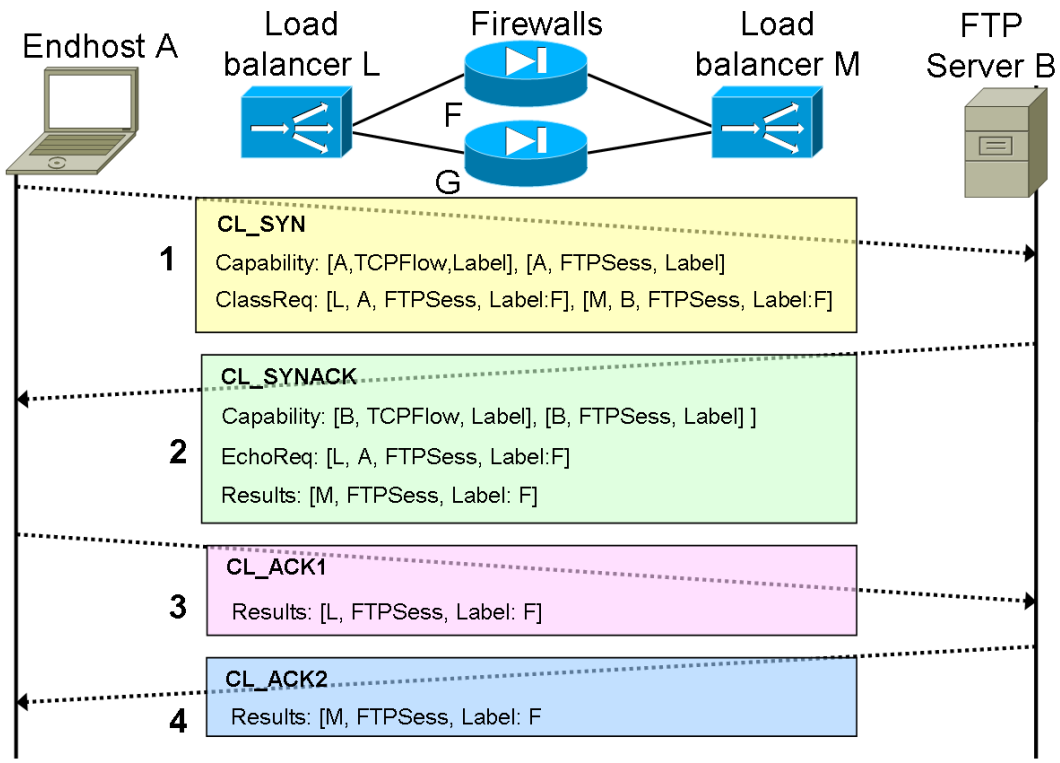

Figure 6.3. Example 3: Coordinating multiple *classifiers* for firewall load balancing.

of an FTP session. Thus, current firewall load balancing solutions are complex both in terms of device implementation as well as in configuration.

GOff simplifies the configuration of firewall load balancing by facilitating explicit coordination between the two load balancers, L and M, in the load balancer pair. It reduces load balancer implementation complexity by offloading the complex operations required for FTP session identification from the load balancers to the endhosts, just like web load balancers offloaded HTTP session identification to endhosts in Section 6.1.3.

Figure 6.3 summarizes the *GOff* signaling associated with firewall load balancing. L adds two *ClassReqs* to A's *CL_SYN* – (i) *ClassReq* c that directs endhost A to label all packets in the *FTPSess* with label F, denoting firewall instance F, and (ii) *ClassReq* c' that directs the final destination to label all packets in the *FTPSess* with same label F. Since L and M are deployed together as pair, L is aware of M and specifies it as the originator of c'. M forwards the *CL_SYN* without adding another *ClassReq*, since it already

contains one with source M. The *CL_SYNACK* sent by B includes the classification result $label : F$ addressed to M, as B acted on the *CL_SYN*'s *ClassReq* c' that was addressed to the final destination. M uses the label to forward the *CL_SYNACK* through the same firewall instance used in the forward direction. The FTP application software at A and B remember the label and include it in all data and control connections in the same FTP session.

A *GOff*-enabled firewall load balancer thus simply reads the label in a packet's *CLayer* header, and forwards it to the firewall instance denoted by the label. Such operational simplicity makes it feasible to integrate firewall load balancing functionality into routers and switches themselves, avoiding the need for expensive special-purpose firewall load balancers.

6.2 Additional Design Issues

This section discusses *CLayer* header internals, signaling robustness, classification correctness, offload feasibility and *GOff* deployability.

6.2.1 Inside the *CLayer* Header

Ideally, the *CLayer* has a free-form, flexible length, key-value format as illustrated in the examples in Section 6.1. However, our proposal is not wedded to any particular header format. Any format that provides the required forwarding performance at the relevant network entities can be used. For example, we may impose a more rigid structure on the *CLayer* to optimize forwarding performance of entities like routers which work faster on fixed length fields. One option is to limit the number of classification results to a small number (say, 3) of fixed length entities. These are placed at the head of the *CLayer* header, thus enabling network entities to easily process them. Since *ClassReqs* and capability lists are infrequent and often associated with heavy-weight processing (for example, web server instance selection), we do not optimize their placement in the header. They can be more loosely encoded – for example, as IP options.

The presence of multiple *classifiers* and *helpers* on a packet's path requires the *ClassReqs* and classification results in a *CLayer* header to be explicitly addressed to specific *helpers* and *classifiers*. Otherwise, multiple *helpers* may respond to the same *ClassReq*, thus wasting resources, or worse, confusing *classifiers* with differing results. Similarly, if multiple *classifiers* request similar classification support, then *helpers* should address the results to those specific *classifiers* to avoid confusion. For example, routers E and C in Figure 6.2 may both request the endhost to label all packets in a particular TCP flow, but with different labels.

We use existing identifiers like IP addresses and MAC addresses to identify *helpers* and *classifiers*. These ids need not be globally routable. They just need to be unique on the path of a particular data flow.

A *CLayer* header contains a unique handle that identifies the *GOff* session to which it belongs. The handle is a concatenation of bits randomly proposed by the two endhosts in the *CL_SYN* and *CL_SYNACK* messages. All *GOff*-related state at *helpers* and *classifiers* is keyed by this handle. In Section 6.2.3, we describe how this handle plays an important role in making *GOff* signaling robust.

6.2.2 Types of Classification Supported

GOff enables standard commonly used classification operations as well as custom ones to be offloaded to *helpers*. In the earlier examples, we discussed various commonly used operations – identifying packets in the same TCP flow or HTTP session and marking them with a specified label. A standardized vocabulary of such commonly used classification operations with well-defined semantics are part of *GOff* specifications and implemented at all relevant nodes. Custom classification operations can be defined out-of-band between co-operating *classifiers* and *helpers* – for example, those within the same administrative domain. These can subsequently be used in standard *GOff* operations. For example, a core router can instruct an edge router to rate limit all packets matching a particular pattern by defining a custom classification action. We hope that the flexibility offered by *GOff* will enable efficient implementation of currently unforeseen classification applications.

GOff is most applicable for classification applications that operate on just the first packet of a flow. Applications that act on collections of packets can still benefit from the classification offload provided by *GOff*. However, *GOff* does not eliminate the packet caching required before the classification decision can be made.

6.2.3 Signaling Robustness

GOff signaling is robust to lost or retransmitted messages, path changes and unexpected state expiration.

Path changes

Changes that do not alter the sequence of *classifiers* and *helpers* on a packet's path have no impact on *GOff* signaling. For example, common local layer-2 and wide area Internet path changes do not affect *GOff* signaling if the switches and routers involved are not *classifiers* or *helpers*.

Path changes involving *classifiers* and *helpers* are detected by the absence of expected classification results, and fixed by *re-signaling*. Figure 6.4 illustrates path changes in the example topology described in Section 6.1.3. To summarize, C and L are *classifiers*; A, W_1 and F are *helpers*; E is simultaneously a *helper* and a *classifier*. In Figure 6.4(a), the network path between endhost A and web server W_1 shifts from edge router F to F'. In Figure 6.4(b), the network path shifts from core router C to C'. In the former case, F' does not insert any classification results addressed to C. In the latter case, classification results are addressed to C, and not C'. Thus, in both cases, the core router (C or C') detects the absence of classification results addressed to it, and initiates re-signaling by setting a *ReSig* flag in the *CLayer* header.

On receiving a *CLayer* header with *ReSig* flag, the endhosts re-run the *GOff* 4-way handshake. The session handle established during original signaling is included in the *CLayer* headers. The endhosts and the *helpers* on the original path use the handle to

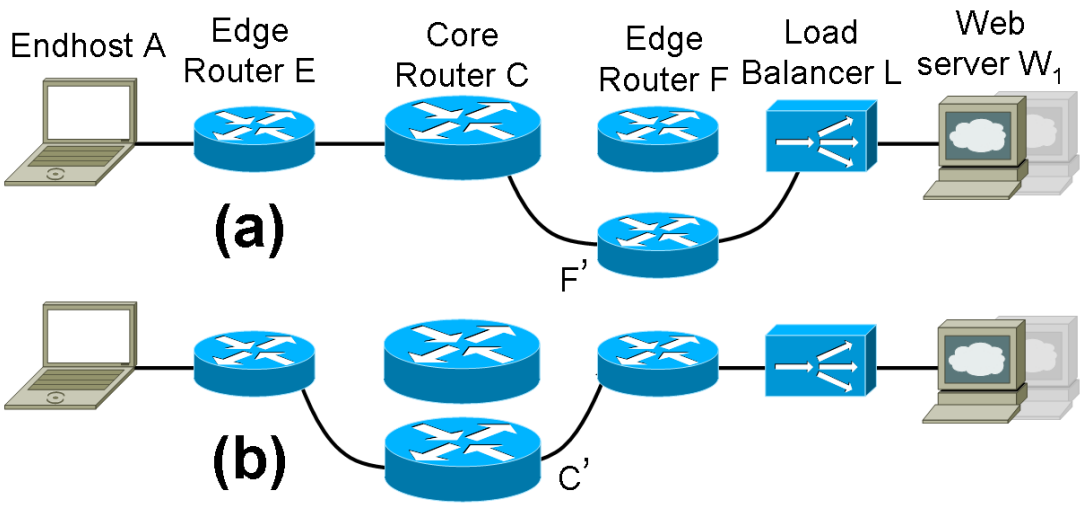

Figure 6.4. Path changes.

retrieve the previously established state, and insert classification results in the re-signaling messages. *Classifiers* append *ClassReqs* only if they do not find the desired classification results addressed to them.

A *classifier* limits the rate at which it sets the *ReSig* flag to avoid re-signaling infinitely when no appropriate *helpers* are present on the new path. A new *helper* that comes online may also pro-actively trigger re-signaling and thus quickly advertise its capabilities to needy *classifiers*. Any duplicate *ReSig* flags received during signaling are ignored.

Most *classifiers* operate correctly during re-signaling. If a *classifier*'s *helpers* are unaffected by the path change, it operates normally without any performance hit. For example, load balancer L continues to choose the correct webserver based on the *Label*:W_1 embedded by *helper* A, irrespective of the path changes in Figure 6.4. Even some *classifiers* with *helpers* affected by the path change function unhindered during re-signaling. For example, core router C can forward packets to IP_A using more expensive regular route lookup.

Some *classifiers* like load balancers may operate incorrectly during re-signaling. For example, if the path changes to include a different load balancer L' that does not understand labels intended for L, packets may be forwarded to the wrong webserver. This is inevitable even in existing load balancer deployments.

Unexpected State Expiration

GOff handles unexpected state expiration at *helpers* and *classifiers* by re-signaling. For example, in Figure 6.4, E may forget its responsibility to label all packets destined to IP_L with x due to timeout or reboot. As in the case of a path change, C initiates re-signaling on receiving a packet without classification results addressed to it. In the common case, re-signaling is necessary only at the start of a new TCP connection in a long-lived session. Such re-signaling incurs little overhead as it is piggybacked on the transport protocol's handshake messages

Classification inaccuracies arise if the new state established as part of re-signaling differs from the original state. For example, if Firefox at endhost A forgets the HTTP session label assigned by the load balancer L, a new web server instance may be chosen by L based on the load conditions during re-signaling. Such session stickiness violation is no different from current scenarios where the HTTP cookie at a web browser expires or is cleared.

Classification soft state established at different *helpers* is often independent of each other (for example, in Sections 6.1.1 and 6.1.3). However, in some scenarios, they are related. In Section 6.1.5, endhosts A and B use the same label so that the firewall load balancer pair can select the same firewall instance in both flow directions. Suppose the state at B expires before A. If A subsequently sends a packet to B, B will not be able to include the correct classification results in its response packet to A. The *GOff* handler at B detects the absence of state identified by the packet's session handle. It runs *GOff* signaling out-of-band to re-establish the missing state before replying to A. Out-of-band signaling can be avoided if A pre-emptively includes the shared *ClassReqs* in the packet if it suspects that B may have forgotten the state – for instance, when sending a new packet after a long gap, or when starting a new TCP connection.

We assume that the session handle is wide enough to avoid collisions on network paths that share common nodes. For example, suppose the handle for an *HTTPSess* is $h_A.h_B$. If B's state is lost, we assume that another endhost C will not propose h_A and B will not reselect h_B before the state keyed by $h_A.h_B$ state at A has expired.

Retransmitted Messages

GOff signaling is naturally resilient to lost packets when piggybacked on reliable transport protocols like TCP. However, special care must be taken to handle retransmissions. Revisiting the example in Section 6.1.3, load balancer L selected the web server instance W_1 on processing the *CL_SYN* from A. Suppose A retransmits the *CL_SYN* (as part of the TCP SYN retransmit) because the *CL_SYNACK* got delayed. If L does not maintain per-flow state, it may assign a different web server instance, say W_2, to the second *CL_SYN*. To prevent confusion, A accepts only the first *CL_SYNACK*. A's TCP stack must also be slightly modified to ensure that any TCP ACK containing a *CL_SYNACK* different from the first one is rejected, as it originated from a different web server instance. To quickly release TCP state at the unused web server instance, A sends a TCP RST with the appropriate classification label embedded in the *CLayer* header.

6.2.4 Correctness of Classification Results

A *helper* may incorrectly classify packets due to malicious intent, buggy implementation or misconfiguration. *GOff* detects and alleviates this problem by imposing structure and semantics on the classification results. However, if detection has very high computation and/or memory requirements, *GOff* is redundant or non-beneficial.

Classifiers like firewalls, which critically depend on the accuracy and integrity of classification results embedded by *helpers*, use labels containing cryptographic hashes in their *ClassReqs* to prevent spoofing. For example, a *GOff*-enabled layer-4 firewall injects a *ClassReq* with the label `H(version, secret, decision, 5-tuple)` into the first packet of a flow (say, TCP SYN). `H` is a one-way hash like MD5 keyed with a `secret` known only to the firewall. Hence, this label unspoofably ties together the accept/drop `decision` for this flow calculated by regular rule lookup with its `5-tuple`. The endhost includes the label in all subsequent packets. The firewall uses it to quickly accept or drop the packet, without performing complex rule lookup [63].

Load balancers also use cryptographically secure labels to prevent a large number of

colluding malicious endhosts from DOSing a particular webserver by spoofing the webserver instance specified in the labels assigned to them. The load balancer imposes the following structure on the label: `[H(src IP prefix/16, server instance, secret), server instance]`. It drops a packet if the hash value `H` calculated with the packet's source IP address prefix does not match the value in the label. However, this approach prevents collusion only between hosts in the same /16. We cannot use the full source IP address in `H` calculation due to the Mega Proxy problem [40, 11] – TCP connections from an endhost may be spread across different NATs in a pool, and thus different TCP connections from the same endhost within the same HTTP session may have different source IP addresses.

Instead of using cryptographic digests, another option is to impart more semantic meaning to the labels. For example, a firewall may embed the *equivalence class* associated with a flow's first packet in the label assigned to it. The equivalence class is a group of packets, characterized by a set of identifying conditions, on which the firewall makes the same decision. Although the total number of equivalence classes can theoretically be exponential in the number of packet fields used in matching, [63] shows that there are much fewer classes in practice. The firewall can easily verify if a packet satisfies the rules associated with the equivalence class in its label, since testing for specific rule matching is much faster than finding the set of matching rules. If the action associated with a packet's equivalence class is `accept` or `drop`, it is immediately forwarded or dropped. However, if its action is `continue`, the firewall performs further fine-grained rule matching on it. Thus, *GOff* helps the firewall quickly jump over a set of rules and narrow down rule matching.

Endhosts are incentivized to co-operate with a *classifier* since their traffic will receive better forwarding performance if they include the label. Packets without labels are subject to regular rule matching or session identification. These packets will be the first to be dropped when the *classifier* is overloaded.

6.2.5 Feasibility of Classification Offload

Classification can be most effectively offloaded to *helpers* if the classification algorithms and rules are not secret and can be easily disseminated to the *helper*. *GOff* allows application developers to trade-off the secrecy and portability of classification rules for performance.

If classification rules are public, a *helper* can perform the whole classification operation by itself, thus greatly lightening the load on the *classifier*. For example, classification associated with HTTP session identification is not secret, and can hence be completely implemented at endhosts. On the other hand, firewall rules are usually kept secret, especially from entities in a different administrative domain. This implies that firewalling cannot be totally offloaded to an end host.

Our *GOff*-based firewall design overcomes rule secrecy restrictions by relying on first packet classification locally at the firewall itself. Endhosts simply reflect the label supplied by the firewall. They are unaware of the filtering rules that derived the label. An additional benefit of this design is that it maintains no per-flow state at the firewall, unlike traditional designs which locally cache rule lookup results for fast application on subsequent packets.

Irrespective of secrecy, classification rules must also be easily disseminated to *helpers* for maximum effectiveness. Keeping a large set of firewall or QoS rules up-to-date at a large number of endhosts, especially over large network distances, is a hard problem. On the other hand, session identification methods in load balancing are mostly standardized and can be implemented in endhosts by default.

6.2.6 Deployability

Deployability of *GOff* depends on the incentives offered to participating entities, the ease with which existing applications can be supported, and the amount of administrative control over the deployment domain (for instance, open Internet versus tightly controlled data center).

Classifiers are incentivized by the new functionality and complexity reduction enabled by *GOff*. Similar to incentives for participation in TCP congestion control, *helpers* are incentivized by the promise of better performance (*i.e.,* higher throughput, lower latency) from their local firewalls, first hop routers or load balancers at websites they visit. However, a *helper* is less incentivized to participate, irrespective of lower performance, when it has a conflict of interest with a *classifier*. For instance, an end host has a conflict of interest with a remote firewall that wishes to drop its packets. Within a single administrative domain, *GOff* adoption may be 'forcefully incentivized', *i.e.,* mandated.

GOff deployability also depends on its ease of implementation and integration into existing networks. *GOff* requires modifying *helpers* and *classifiers*. The close similarity between the *GOff* and BSD socket libraries and the small number of lines to port existing applications demonstrate that *GOff* can be easily integrated into existing applications. *GOff* daemon functionality can be embedded in future OS versions or can be installed as a standalone system service (as described in Section 6.3).

An Internet-wide *GOff* deployment does not require a fork-lift upgrade of the entire network. *GOff* traffic can co-exist with non-*GOff* traffic, and only entities wishing to benefit from the *CLayer* need to be upgraded. However, a *GOff*-enabled connection between two endhosts requires all *classifiers* and forwarding elements on the path between them to be *GOff*-aware or to ignore the *CLayer* headers and forward packets unmodified. For example, layer-2 switches and simple layer-3 routers simply forward *GOff* packets unmodified even though they are not *GOff*-aware. However, a non-*GOff*-aware layer-4 router or firewall may consider *GOff* packets as malformed/suspicious and thereby drop them. Such drops are possible even if the *CLayer* header is tucked into an IP option [57] rather than implemented as a separate layer. Hence, an endhost can open a *GOff*-enabled connection with another endhost only after running a path discovery protocol to ascertain *GOff* support.

A data center network is a more ideal candidate for *GOff* deployment than the wide area Internet. The single administrative domain enables us to easily modify endhosts, as advocated by other new data center network architectures like Monsoon [61]. Moreover, the multiple new data centers being built today [22] offer hope for a cleanslate *GOff* imple-

mentation. A *proxy* at the data center ingress can cleanly separate the data center network from the Internet, and adds the appropriate *CLayer* headers. The implementation of such a proxy that can scale to data center workloads is an open challenge. This approach however confines *GOff* functionality and benefits to within the data center. For example, we cannot offload HTTP session identification to endhosts in the Internet.

6.3 Implementation

We prototyped *GOff* using Click [72] and *GOff*-enabled a variety of *helper* and *classifier* applications. Table 6.1 lists the line count for the core *GOff* code (C++) and extra lines (C/C++) required to port existing applications.

The *GOff* implementation at a *helper* node consists of two parts – a daemon and a network socket library. The daemon implements the core *GOff* functionality, *i.e.,* control plane signaling and data plane classification. *Helper* applications interact with the daemon through *GOff* socket library calls, as described in Section 6.1.2. In an ideal clean slate implementation, the functionality implemented by the daemon will be part of the OS network stack and the socket library will be direct system calls. However, in our prototype, the daemon is a userlevel Click router, with which the socket library interacts over a local TCP connection. The daemon uses the *tun* [93] device to intercept outgoing packets and to transfer incoming packets to regular network processing after stripping out *CLayer* headers.

GOff-enabling an existing *helper* application often simply involves replacing BSD socket calls with their *GOff* equivalents. For example, porting *wget* required changes in just 10 lines of code. *GOff*-enabling applications like the *elinks* web browser, which support richer semantics like HTTP sessions consisting of multiple TCP flows, requires slightly more work. The application must remember the session handle for inclusion in **clconnect** calls for all TCP connections in the same session. Often, as in the case of *elinks*, we only need to augment existing session data structures in the application code with an extra field to store the *GOff* session handle.

Table 6.1. Lines of code of our prototype *GOff* implementation

Component	Lines of code
elinks text-based web browser	40
lighttpd web server	19
httperf HTTP benchmark tool	7
ncftp FTP client	38
oftp FTP server	45
wget command line HTTP client	10
nuttcp TCP throughput benchmark tool	13
haproxy [14] layer-7 load balancer	85
QoS & MPLS router	111
Layer-4 firewall	308
Firewall load balancer	153
Layer-4 lb	190
GOff socket library & daemon	4025

Our prototype *GOff* implementation also supports unmodified *helper* applications, since its core functionality is implemented by the standalone daemon. However, classification semantics are restricted to those the daemon can infer without application input. For example, it cannot aggregate TCP connections into HTTP sessions. However, it can classify all packets originating from the same host. Such information can be used by simple layer-3 load balancers to correctly spread traffic. Unlike using source IP addresses, this method does not violate HTTP session stickiness nor causes load skew in the presence of NATs or Mega Proxies.

GOff processing in a layer-4 *classifier* like a QoS router mainly involves adding *ClassReqs*, extracting *CLayer* headers from packets and using the embedded classification results to appropriately forward them. Using libraries from our core *GOff* code, we *GOff*-enabled existing Click modules implementing different *classifier* functionality in under 200 lines of code on average. *GOff*-enabling the *haproxy* layer-7 load balancer was similar to that of a *helper* application, albeit using some additional *GOff* socket calls to retrieve classification results from the daemon.

We used *Google protobufs* [13] to encode/decode *CLayer* headers, instead of designing our own header format. Although the dynamic nature of *protobufs* slightly increases header

size and encode/decode complexity over a fixed width format, it greatly increased the pace and ease of prototyping.

6.4 Evaluation

So far, we qualitatively demonstrated the flexibility and reduction in classifier implementation and configuration complexity provided by *GOff*. In summary, simultaneously supporting a variety of applications (such as forwarding, load balancing, firewalling, and QoS) attests for flexibility and generality. By offloading classification operations to entities with better semantic context, *GOff* reduces the implementation complexity of classification applications. For example, *GOff* simplifies firewall and web load balancer implementation by offloading complex operations like FTP and HTTP session identification to endhosts, thus avoiding expensive deep packet inspection. By providing a single generic mechanism for classification offload and by enabling explicit coordination between *helpers* and *classifiers*, *GOff* also simplifies configuration.

In this section, we quantitatively demonstrate how *GOff* improves the scalability of classification applications using firewalling and load balancing as examples. The firewalling example further illustrates how *GOff* spurs classification offload in traditionally centralized applications, and thereby improves performance. Before describing the firewalling and load balancing results, we first micro-benchmark our prototype *GOff* implementation used for the experiments. We conclude this section by summarizing *GOff*'s limitations.

6.4.1 *Helpers*

GOff introduces a small processing overhead at *helpers* – under one microsecond/packet on average in our prototype implementation. We counted the CPU ticks taken by different *GOff* operations using the RDTSC x86 instruction on a 3GHz desktop PC running Linux (10000 ticks $\approx$ 3.34 microseconds). Table 6.2 summarizes the results. The *FromApp* phase covers the time between packet capture from the application till it is forwarded out

Table 6.2. *GOff* processing overheads at *helpers*

Phase	Min Ticks	Avg Ticks	Std. Dev	Count
FromApp	1464	2564	2477	200879
ToApp	2104	2918	2742	200879

of the appropriate network interface. The main operations in this phase are retrieving the appropriate session state, performing classification and embedding a *GOff* header with classification results into the packet. The *ToApp phase* covers the time between receipt of a packet from an external network interface till it is handed off to the application through the `tun` device [93]. The major operations in this phase are updating session state and removing *CLayer* headers.

Our throughput measurement experiments indicate that packet capture using the `tun` device is a significant overhead in our userlevel prototype implementation. In the absence of both packet capture and *GOff* processing, `nuttcp` [29] measured a TCP throughput of 941 Mbps between two PCs A and B. The throughput drastically dropped to 635 Mbps when the experiment was conducted using packet capture, without any *GOff* processing. This throughput drop is solely an artifact of our userlevel software prototype implementation, and not an inherent limitation of *GOff*. Addition of *GOff* processing decreased the throughput to 536 Mbps, an overhead of only 16% over the baseline 635 Mbps.

We believe that a kernel implementation of *GOff* will avoid the packet capture overhead. Furthermore, by pre-allocating extra per-packet buffer space for *CLayer* headers, the expensive packet copies that currently slow down our userlevel *GOff* implementation can be avoided.

6.4.2 Packet Overheads

A *CLayer* header containing a single classification result with a 16-byte label occupies 47 bytes, in our prototype implementation that uses Google protobuf's dynamic packet format. In protobufs, each field is prefixed with a 1-byte tag and some fields with an additional length field. Moving to a fixed header format saves 10 bytes.

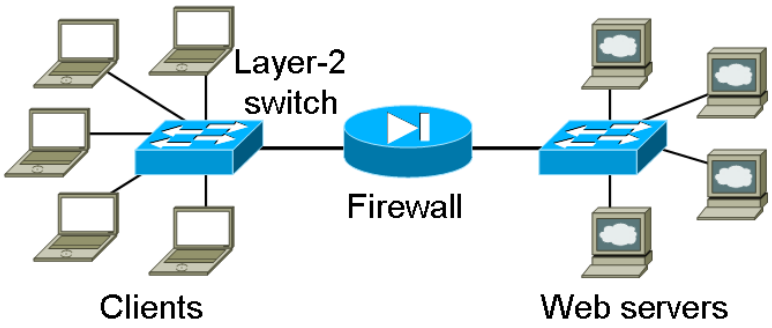

Figure 6.5. Topology used for firewall throughput measurement.

Smaller labels can be used for applications more tolerant to spoofing by *helpers* (for example, QoS and load balancing). The label can be as small as one byte, instead of the 16 bytes needed for storing a cryptographic hash. Further space savings can be achieved by shrinking the 8-byte session handle to 4 bytes, and the 4-byte source and destination identifiers (*i.e.,* IP addresses) of a classification result to 2 bytes each. Such reduction in handle or identifier size is unlikely to result in a collision as uniqueness is required only within individual data paths. After incorporating the size reductions described above, the minimum data packet overhead for one classification result is 14 bytes.

6.4.3 Firewalling

The throughput of a regular firewall decreases with increasing rule set size [97]. Our *GOff*-enabled firewall scalably maintains constant throughput that is two to four times that of a regular firewall at large rule sizes

Figure 6.5 shows our experimental topology, created on the DETER [47] testbed. We used the Click [72] `IPFilter` module as the base of our regular and *GOff*-enabled firewalls. Firewall throughput was calculated as the sum of throughputs achieved by simultaneous large file transfers between `wget` clients and `lighttpd` servers.

We used three different rules sets at the firewall:

1. `Snort4` : Port number matches were sampled from the over 600 unique rule headers

(*i.e.,* involving just packet 5-tuples) in the Snort rule set [37] [98]. Due to lack of IP diversity in the Snort rule set, source and destination IP matches were randomly drawn from a pool of 4 prefixes.

2. `Snort250` : Same as Snort4, but source and destination IP matches were drawn from a pool of 250 prefixes.

3. `RandomIP` : Source and destination IP matches were drawn from a random pool of 100 prefixes. Rules ignored port numbers.

For each rule set, we ensured that the rule matching our file transfer traffic was last. This enabled us to measure worst case performance, independent of the traffic mix.

Figure 6.6 shows the throughput drop of the regular firewall as the rule set size increases from 100 to 4000. For the `Snort4` and `Snort250` sets, the throughput drops more than 80% – ≈77MB/s → ≈12MB/s. For the `RandomIP` set, it drops 68% – ≈87MB/s to ≈27MB/s. The *GOff*-enabled firewall (line `Snort4.clayer`) maintains a constant throughput of ≈50MB/s, irrespective of rule set characteristics and size. Only ruleset `Snort4` is shown for the *GOff* firewall to avoid clutter.

A *GOff*-enabled firewall thus outperforms a regular firewall when the rule set size is above an *attractiveness threshold*. More importantly, it sustains a constant throughput even as the rule set size increases, thus demonstrating good scalability. In our experiments, the attractiveness threshold is 500 for the `Snort` rule sets and 1200 for the `RandomIP` rule set. Many firewall deployments already have rule sets that are larger than our thresholds. A 2004 study [95] found that firewalls have upto 2671 rules. The biggest classifier in [63] had 1733 rules, while the biggest edge router ACL set in [86] had 4740 rules. We expect rule set size to continue to grow as the size and complexity of networks increase. Thus, the attractiveness of a *GOff*-enabled firewall increases over time, even if the attractiveness threshold is higher than that in our experiments.

Our experiments indicate opportunity for increasing throughput and lowering the attractiveness threshold by optimizing the firewall implementation. Line `Snort4.clayer.nohash` in Figure 6.6 shows that skipping the hash computation increases throughput by over 30%

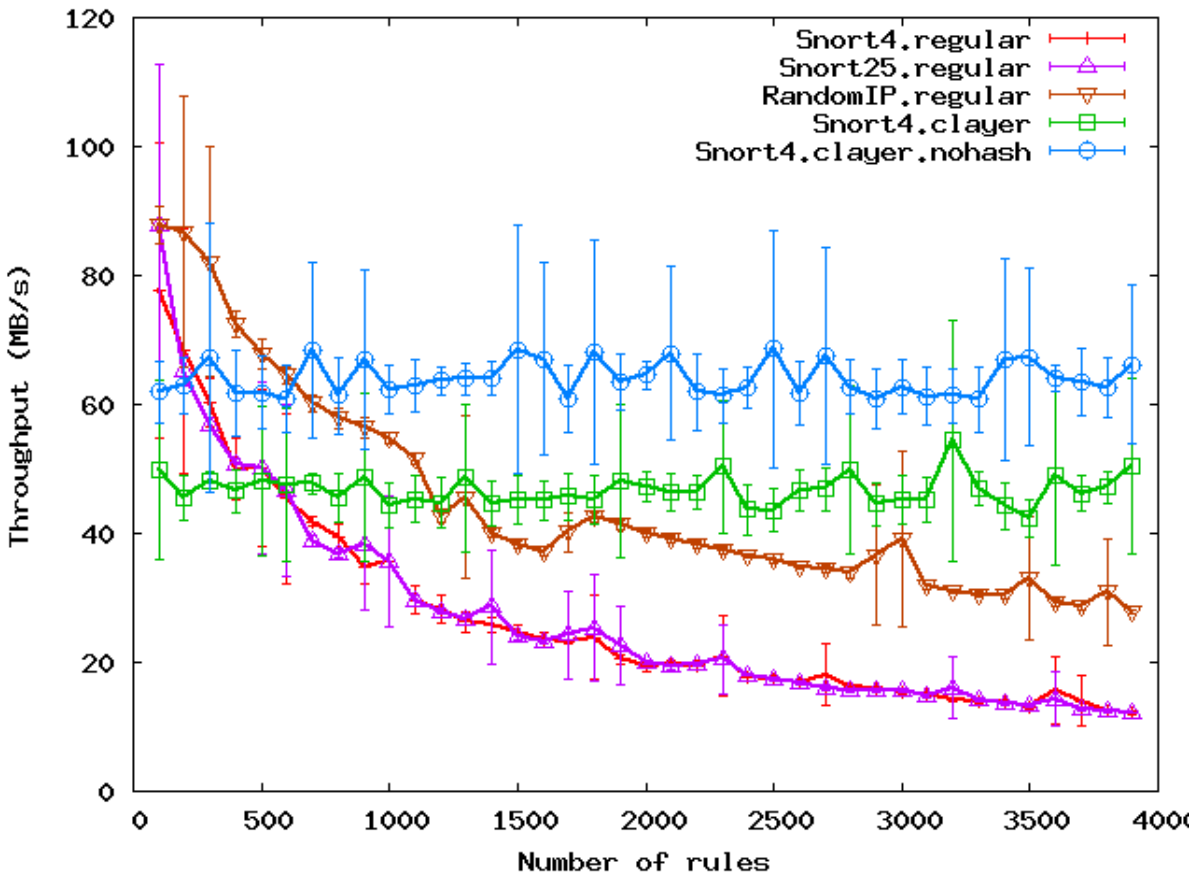

Figure 6.6. Firewall throughput versus number of rules.

and lowers the attractiveness thresholds to 300 and 600 for the `Snort` and `RandomIP` rule sets. Thus, optimizing the cryptographic hash computation (say, with hardware acceleration) offers great potential for improving performance and lowering the threshold.

6.4.4 Load Balancing

Our *GOff*-enabled layer-4 load balancer prototype attained 68% more connections/second and 63% more throughput than the `haproxy` layer-7 web load balancer, while providing similar HTTP session stickiness. We used `httperf` and parallel `wget` to benchmark the load balancers. Our layer-4 load balancer supported 4448 connections/second and 59 MB/s

throughput, while `haproxy` supported 2646 connections/second and 36 MB/s throughput. To factor out the overhead due to packet capture used by our prototype *GOff* implementation, we employed plain packet capture (*i.e.,* without *GOff* processing) in our `haproxy` experiments.

Unlike our layer-4 load balancer, our *GOff*-enabled layer-7 `haproxy` (*viz.*, `clhaproxy`) performed worse than a regular `haproxy` – 2090 connections/second, 32.5 MB/s versus 2646 connections/second, 36 MB/s. The main reason for `clhaproxy`'s poor performance compared to `haproxy` is the packet copy overhead incurred as part of inserting and removing *CLayer* headers from each packet. Just like regular `haproxy`, `clhaproxy` acts as a TCP endpoint for client TCP connections and opens new TCP connections to web servers. It thus does not have the luxury of avoiding expensive TCP processing like in a layer-4 load balancer.

We believe that a kernel *GOff* implementation that avoids expensive packet copies will take `clhaproxy` performance beyond that of `haproxy`. However, if a layer-7 load balancer anyway performs TCP stream reconstruction and HTTP parsing for advanced applications like URL based content blocking, regular HTTP cookies can be used for session identification as well. Although *GOff* does not offer significant performance improvements in this particular scenario, it offers the convenience of having a single mechanism to simultaneously implement and configure all deployed classification applications.

6.4.5 Limitations

In this section, we summarize the limitations of *GOff* explained in earlier sections:

1. *GOff* incurs per-packet space and processing overheads. However, performance benefits due to complexity reduction enabled by *GOff* often mask the impact of these overheads.

2. All classification applications do not equally benefit from *GOff*. *GOff* offers only limited benefits to:

(a) applications where classification result accuracy is critical but very expensive to check,

(b) applications which anyway perform complex operations for some other purpose but may be overloaded for classification, and

(c) applications where no *helpers* exist or are all disincentivized to participate.

However, the overall reduction in configuration complexity enabled by *GOff* still benefits the network.

3. Although *GOff* is incrementally deployable and offers the familiar BSD sockets API, it requires modifications to endhosts and other entities that wish to benefit from *GOff* functionality.

4. Like in existing mechanisms, drastic path changes or memory losses at *helpers* may result in dropped packets or misclassification. For instance, a load balancer may forward the requests of a client which forgot the HTTP cookie to the wrong webserver instance.

6.5 Related Work

Many prior works advocate distributing packet classification load across various network entities. Unlike the generic multi-application approach we describe in this paper, the mechanisms proposed by these works are point, and often ad-hoc, solutions focusing on a particular type of classification application and a specific layer.

Packet forwarding: In MPLS [25], label switch routers in the network core offload expensive route lookup to edge label switch routers. Edge routers classify packets based on their destination IP (*i.e.,* route lookup) and inject labels into them to mark the results. Downstream routers avoid reclassification by simply using these labels in their forwarding decisions. Ipsilon [18] flow-switching is more general than its successor MPLS, and supports packet classification at the network and transport layers. Out-of-band IFMP (Ipsilon Flow Management Protocol) allows adjacent nodes to configure the labels to be used for different

flows. *GOff* supports classification across layers 2 to 7, and uses an in-band signaling protocol not restricted to adjacent nodes.

QoS: In Diffserv [2], endhosts, or more frequently first hop routers, classify packets and record the desired QoS in the DS field of the IP header. In CSFQ [90], edge routers label packets of a flow based on its flow rate. The 20-bit *flowid* IPv6 header field [19] provides a mechanism for end-hosts to uniquely identify a flow with any desired semantics. Routers in the network core can provide differential QoS to packets based on their DS fields, labels or flow-ids without performing expensive reclassification.

Session Identification: The OSI network stack [100] includes a *session layer* to capture application session semantics. Although originally designed for offloading webserver state to endhosts, HTTP cookies [15] are widely overloaded as a means to identify multiple TCP flows in an HTTP session. Unlike HTTP cookies and the OSI session layer, *GOff* is not restricted to the application layer – it works across layers 2 to 7. In addition, *GOff* makes the session id available to a load balancer in an easily readable packet header location, as opposed to deeply buried inside application headers.

Some prior works adopt an extreme approach of moving the entire application requiring packet classification to endhosts. In Distributed firewalls [48], an endhost matches a packet against the firewall rules, and independently decides to accept or drop it. Network Exception Handlers [69] offload traffic shaping to endhosts by supplying them with the network topology and notifying them about events like link failures. The endhost classifies packets and drops or rate limits them as specified by their exception handlers. Our work targets the more conventional and widely deployed scenario where an in-network entity (like a router or middlebox) is involved (often in a critical role) in implementing the functionality that requires packet classification. *GOff* can be used to communicate the results of network exception handlers to on-path entities.

In Ethane [51], packet classification is offloaded to a centralized controller. Based on the initial packets of a transport flow, the controller classifies packets and installs forwarding table entries at switches on the flow's network path. Thus, unlike in *GOff*, classification

in Ethane operates at the granularity of transport flows. *GOff* enables packet classification by endhosts and on-path entities like middleboxes, which may often be better suited for a particular type of packet classification than the centralized controller. Moreover, *GOff*'s distributed approach avoids a classification choke point and a centralized point of failure. *GOff* trades off per-packet overhead for the overhead of forwarding table state establishment at flow startup.

RFC 3697 [19] advocates that the IPv6 *flowid* header field targeted at QoS routers can be filled in by endhosts as they are best suited to identify a flow. They can thus be extended to applications beyond QoS, for example, session identification in web load balancing. However, unlike *GOff*, it supports only one application at a time, and does not provide any signaling mechanism to inform/configure the entities that use the flowid field.

COPS [71] proposes *iBoxes* that classify a packet using deep packet inspection and then summarize the results in an *annotation layer* within the packet. A packet's annotation layer influences the forwarding decision (forward, drop, rate limit) at subsequent *iBoxes* it traverses. The annotation layer doubles up as an in-band management plane to control the *iBoxes* in the network. *GOff* simultaneously supports a variety of classification applications, in addition to security. Instead of a management plane, *GOff* includes a generic protocol for configuring classification offload.

CLayer headers are similar to X-trace [56] annotations in that their semantics and purpose can span multiple protocol layers. X-trace annotations contain metadata for reconstructing an application request's path through a distributed system to aid network diagnostics. In contrast, *CLayer* headers act as a unified scratchpad that carries signaling messages of our distributed offload protocol, and classification results with varying semantics embedded by different *helpers*.

Active networking [92] enables endhosts to customize network functionality by embedding code within packets to be executed at routers and programmable network elements. *CLayer* headers carry non-executable opaque bags of bits whose semantics depend on the classification application to which they are directed. This more restrictive nature of *GOff*

avoids the security risks of executing untrusted code, while still enabling endhosts to influence the fate of their packets within the network.

GOff signaling component borrows ideas from protocols used in different applications, in addition to HTTP cookies. Explicit Congestion Notification [38] is an inband signaling mechanism used by routers to instruct endhosts to reduce their packet sending rate. MIDCOM [23] is a signaling protocol by which endhosts configure middleboxes – for example, instruct a NAT to open a specific port. The RSVP [42] resource reservation protocol, commonly used in conjunction with Intserv, allows an endhost to set up filter specs at routers to aid classification and differential treatment of packets it originates. In capabilities based DOS attack protection mechanisms [96], a destination echoes back markings made on capability request packets by enroute routers back to the sender. The sender includes these markings in the subsequent data packets it sends to the destination.

Stateful Distributed Interposition (SDI) [84] and Causeway [52] provide mechanisms to automatically propagate and share contextual information and metadata across tiers of a multi-tier system or within different layers in an OS. OS-level support for SDI or Causeway obviates the need to modify endhosts to maintain session information and embed *CLayer* headers. This simplifies *GOff* implementation and deployability. *GOff* simply becomes a 'meta-application' of SDI and Causeway with the explicit goal of enhancing packet classification.

6.6 Summary

In this chapter, we demonstrated how treating packet classification as a fundamental network primitive via *GOff* reduces the implementation and configuration complexity of packet classification, while improving flexibility, performance and scalability. Inspired by the variety of existing applications we implemented using *GOff*, we believe that *GOff* is flexible enough to support future classification applications, without inventing additional point solutions.

Chapter 7

Conclusions & Future Directions

Packet classification is a key component of many different network operations. In spite of its ubiquity, it is not treated as a fundamental network primitive. The various existing classifier deployment and classification placement solutions are inflexible, inefficient and hard to configure. In this thesis, we addressed these three main limitations of today's packet classification solutions.

The main contribution of this thesis is to *elevate packet classification as a fundamental network primitive*. To do so, we defined a new *classification layer* (*CLayer*, in short) in the protocol stack and a pair of associated control plane protocols – *policy-based classifier deployment* and *generic classification offload*. We also developed a *classifier* model to better understand the configuration requirements and operational behaviors of different classifiers.

Together with the *CLayer*, policy-based classifier deployment, generic classification offload and the classifier model enable us to address today's packet classification challenges in a holistic manner. Next, we summarize their functionality and benefits. We then present future research directions.

7.1 Policy-based Classifier Deployment

Policy-based classifier deployment simplifies the process of deploying and configuring classifiers. A network administrator specifies policies dictating the sequence of classifiers that must process different types of traffic. The underlying policy-aware forwarding layer enforces these policies by explicitly redirecting traffic to the appropriate classifiers. Such explicit indirection guarantees correct classifier traversal under all network churn conditions. Policy-based specification simplifies network configuration by avoiding the need to overload path selection mechanisms to enforce the desired classifier sequence. In addition, it improves network flexibility – changing the classifier traversal sequence requires a simple policy change. It also improves efficiency by ensuring that traffic only traverses classifiers required by policy, and by load balancing traffic across all available classifier boxes.

We have demonstrated the benefits of policy-based classifier deployment using our prototype implementation of the *policy-aware switching layer* (*PLayer*, in short). The *PLayer* is an instantiation of policy-based classifier deployment tailored to data center and enterprise networks, that requires no changes to existing endhosts and classifiers.

7.2 Generic Classification Offload

Generic classification offload provides a single mechanism to distribute classification load across different entities in the network. It reduces network complexity by avoiding the need to separately implement and configure diverse mechanisms like MPLS, Diffserv and HTTP cookies for different classification applications. By leveraging *CLayer* headers to carry classification hints, it avoids the deep packet inspection efficiencies incurred by current classification offload solutions like HTTP cookies. Its generic design provides the flexibility to easily support classification offload in new or traditionally centralized classification applications.

We have demonstrated the benefits of generic classification offload using *GOff*, a specific instantiation of generic classification offload. Our prototype *GOff* implementation could

simultaneously support existing classification applications such as QoS, route lookup, and HTTP session identification. Our *GOff*-enabled HTTP layer-4 load balancer achieved 68% more connections/second and 63% more throughput than a regular layer-7 HTTP load balancer, while providing similar HTTP session stickiness semantics. We were also easily able to provide the benefits of classification offload to firewalling, a traditionally centralized classification application. Our *GOff*-enabled firewall scalably sustained throughput that was two to four times higher than that of a regular firewall at large rule sizes.

7.3 Classifier Model

The *classifier model* provides a succinct and clear language to describe how different classifiers process packets. We represented many different classifiers like firewalls and load balancers using the model. The clear understanding of classifier behavior provided by the model helps network administrators plan and troubleshoot classifier deployments. In addition, it helps network researchers understand how different classifier behavior interact with the artifacts of their research. We have developed tools to semi-automatically infer and validate models by observing classifier operations.

7.4 Future Research Directions

The *CLayer* opens up a new perspective of thinking about the role of classification in network architectures. Various network operations are simply specific instantiations of classification that are configured by different control protocols. For example, layer-3 packet forwarding can be viewed as packet classification configured by protocols like BGP and OSPF. At a high level, exploring how various network operations can be modeled as classification and how the *CLayer* can help support them is an interesting research goal. In this final section of this thesis, we describe some specific future research directions that can contribute towards this goal.

7.4.1 Integrating with New Internet Architectures

Many recently proposed new Internet architectures such as Internet Indirection Infrastructure (*i3*) [88] and Data Oriented Network Architecture [73] provide packet forwarding and security functionality that is different from the currently deployed IPv4-based Internet architecture. *CLayer* functionality is orthogonal to the functionality provided by these architectures. Adding *CLayer* functionality into many diverse new Internet architectures will be an interesting challenge.

7.4.2 *CLayer* Deployment

An Internet-wide deployment of *CLayer* and its associated protocols is very hard to achieve. However, the many data centers currently under construction provide a good opportunity for demonstrating the benefits of the *CLayer*. Data centers are ideal for deploying *CLayer* and its associated policy-based deployment and generic offload because all entities within it are under a single administrative domain.

Our prototype implementation of the policy-aware switching layer (*PLayer*) provides a policy-based classifier deployment solution tailored for layer-2 data center networks. Expanding *PLayer* support to mixed layer-2/layer-3 or layer-3 data center networks has not been addressed yet. The current *PLayer* design and implementation does not require any modifications to endhosts and classifiers. Another future research direction is to explore the additional benefits that are possible if endhosts and classifiers are modified. For example, classifier modifications can simplify *PLayer* mechanisms to handle policy and middlebox churn.

Our current *GOff* implementation relies on *GOff* support at both endpoints. In a data center, we have administrative control over the servers and internal network entities, but not over external clients. This implies that our current *GOff* prototype is not suitable for a data center environment. To address this limitation, we need a *GOff proxy* that sits at the data center ingress and implements *GOff* signaling on behalf of external clients. The large

number of external clients and complex applications served by current data centers make designing a scalable *GOff* proxy a challenging research task.

7.4.3 Prototyping in Hardware

We implemented *PLayer* and *GOff* in software for ease and speed of prototyping. However, our software prototypes cannot match the speed of hardware based solutions. Fortunately, programmable network hardware is gaining popularity. FPGA based network platforms such as NetFPGA [27], and programmable switch fabrics from chipset vendors like Broadcom [8] are now available to researchers. Mixed software-hardware platforms like OpenFlow [30] enable the customization of hardware forwarding functionality through software.

Prototyping the *PLayer* and *GOff* on these platforms and evaluating their performance will be an interesting research exercise. The main challenges here are the complexity of hardware programming and the limitations of restrictive software programming interfaces.

7.4.4 Simplifying Policy Specification and Validation

Policy-based classifier deployment simplifies network configuration. However, a network administrator must still correctly specify the policies. In complex networks containing many different types of classifiers and traffic, policies can grow very large and complex.

We have developed a basic graphical user interface to construct policies and statically validate their correctness. Enhancing our interface with the ability to intelligently suggest policies and flag missing ones based on well-known best practices will be an interesting activity. Dynamically validating the accuracy and efficacy of policies is another challenging research direction.

7.4.5 Dynamic Classification Offload

Our generic classification offload protocol currently enables classification offload only to entities that are already upstream on the network path. This is not an intrinsic limitation of generic classification offload. In some scenarios, it may be beneficial to offload classification to entities off the normal network path. For example, a firewall may temporarily offload classification decisions on suspicious traffic to a more powerful offpath intrusion detection box. If the intrusion detection box flags the traffic as normal, the firewall can directly classify subsequent packets by itself.

Implementing such dynamic classification offload requires close coordination between policy-based classifier deployment and generic classification offload. Designing a scalable and time-sensitive protocol that enables such fine-grained coordination is a challenging task. Exploring how new architectures like NOX [62] may be used to dynamically change the path of traffic is also an interesting research direction.

7.4.6 A Repository of Classifier Models

We have manually constructed model instances of commonly available software load balancers and firewalls. We hope that classifier device vendors, network administrators and researchers will construct model instances for more classifiers, especially hardware ones, and contribute them to a publicly available model repository. Models downloaded from the repository can be used to validate observed classifier behavior.

Automated model inference and validation tools will greatly enhance the utility of our model. Blackbox testing, natural language analysis of classifier documentation and analysis of classifier source code may aid the development of such tools. The diversity and complexity of classifier behavior make the development of such tools a challenging research activity.

Bibliography

[1] A Border Gateway Protocol 4 (BGP-4). RFC 1771.

[2] An Architecture for Differentiated Services. RFC 2475.

[3] Architecture Brief: Using Cisco Catalyst 6500 and Cisco Nexus 7000 Series Switching Technology in Data Center Networks. http://www.cisco.com/en/US/prod/collateral/switches/ps9441/ps9402/ps9512/White_Paper_C17-449427.pdf.

[4] BalanceNG: The Software Load Balancer. http://www.inlab.de/balanceng.

[5] Behavior Engineering for Hindrance Avoidance. http://www.ietf.org/html.charters/behave-charter.html.

[6] Beth Israel Deaconess Medical Center. Network Outage Information. http://home.caregroup.org/templatesnew/departments/BID/network_outage/.

[7] BladeLogic Sets Standard for Data Center Automation and Provides Foundation for Utility Computing with Operations Manager Version 5. Business Wire, Sept 15, 2003, http://findarticles.com/p/articles/mi_m0EIN/is_2003_Sept_15/ai_107753392/pg_2.

[8] Broadcom. http://www.broadcom.com/products/Enterprise-Networking.

[9] Cisco catalyst 6500 series switches solution. http://www.cisco.com/en/US/products/sw/iosswrel/ps1830/products_feature_guide09186a008008790d.html.

[10] Cisco Systems, Inc. Spanning Tree Protocol Problems and Related Design Considerations. http://www.cisco.com/warp/public/473/16.html.

[11] Deployment Issues. http://technet.microsoft.com/en-us/library/cc783135.aspx.

[12] F5 Application Delivery Controller Performance Report, 2007. http://www.f5.com/pdf/reports/f5-performance-report.pdf.

[13] Google Protocol Buffers. http://code.google.com/p/protobuf/.

[14] HAProxy. http://haproxy.1wt.eu.

[15] HTTP State Management Mechanism. RFC 2109.

[16] IEEE 802.1Q-2005 - Virtual Bridged Local Area Networks. http://standards.ieee.org/getieee802/download/802.1Q-2005.pdf.

[17] IEEE EtherType Field Registration Authority. https://standards.ieee.org/regauth/ethertype/.

[18] Ipsilon Flow Management Protocol Specification for IP. RFC 1953.

[19] IPv6 Flow Label Specification. RFC 3697.

[20] ISL & DISL: Cisco Inter-Switch Link Protocol and Dynamic ISL Protocol. http://www.javvin.com/protocolISL.html.

[21] LDP Specification. RFC 3036.

[22] Microsoft: Datacenter Growth Defies Moore's Law. InfoWorld, April 18, 2007, http://www.pcworld.com/article/id,130921/article.html.

[23] Middlebox Communication Architecture and Framework. RFC 3303.

[24] Middleboxes: Taxonomy and Issues. RFC 3234.

[25] Multiprotocol Label Switching Architecture. RFC 3031.

[26] Net-SNMP. http://net-snmp.sourceforge.net.

[27] NetFPGA. http://netfpga.org.

[28] Netgear MR814 Router. http://kbserver.netgear.com/products/mr814.asp.

[29] nuttcp. http://linux.die.net/man/8/nuttcp.

[30] OpenFlow. http://www.openflowswitch.org.

[31] OSPF Version 2. RFC 2328.

[32] Overview and Principles of Internet Traffic Engineering. RFC 3272.

[33] Personal communications with Maurizio Portolani, Cisco.

[34] Policy based routing. http://www.cisco.com/warp/public/732/Tech/plicy_wp.htm.

[35] Ruby on Rails. http://www.rubyonrails.org.

[36] Sarbanes-Oxley Act 2002. http://www.soxlaw.com/.

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

[38] The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168.

[39] The IP Network Address Translator (NAT). RFC 1631.

[40] The Mega-proxy Problem for e-Commerce Providers. http://www.securitytechnet.com/resource/rsc-center/vendor-wp/intel/mega-proxy_NP1667.pdf.

[41] The netfilter.org project. http://netfilter.org.

[42] The Use of RSVP with IETF Integrated Services. RFC 2210.

[43] US Search Engine Rankings, September 2007. http://searchenginewatch.com/showPage.html?page=3627654.

[44] Cisco data center infrastructure 2.1 design guide, 2006.

[45] Allman, M. On the performance of middleboxes. In *IMC 2003*.

[46] Arregoces, M., and Portolani, M. *Data Center Fundamentals*. Cisco Press, 2003.

[47] Bajcsy, R., Benzel, T., Bishop, M., Braden, B., Brodley, C., Fahmy, S., Floyd, S., Hardaker, W., Joseph, A., Kesidis, G., Levitt, K., Lindell, B., Liu, P., Miller, D., Mundy, R., Neuman, C., Ostrenga, R., Paxson, V., Porras, P., Rosenberg, C., Tygar, J. D., Sastry, S., Sterne, D., and Wu, S. F. Cyber defense technology networking and evaluation. *Commun. ACM 47*, 3 (2004 `http://deterlab.net`), 58–61.

[48] Bellovin, S. M. Distributed firewalls. *;login: 24*, Security (November 1999).

[49] Caesar, M., Caldwell, D., Feamster, N., Rexford, J., Shaikh, A., and van der Merwe, J. Design and Implementation of a Routing Control Platform. In *NSDI 2005*.

[50] Caldwell, D., Gilbert, A., Gottlieb, J., Greenberg, A., Hjalmtysson, G., and Rexford, J. The Cutting EDGE of IP Router Configuration. In *HotNets 2003*.

[51] Casado, M., Freedman, M. J., Pettit, J., Luo, J., McKeown, N., and Shenker, S. Ethane: Taking Control of the Enterprise. In *SIGCOMM 2007*.

[52] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W. Causeway: operating system support for controlling and analyzing the execution of distributed programs. In *HOTOS 2005*.

[53] Demmer, M., Fall, K., Koponen, T., and Shenker, S. Towards a Modern Communication API. In *HotNets 2007*.

[54] Elmeleegy, K., Cox, A., and Ng, T. On Count-to-Infinity Induced Forwarding Loops Ethernet Networks. In *Infocom 2006*.

[55] Elmeleegy, K., Cox, A. L., and Ng, T. S. E. "etherfuse: an ethernet watchdog". In *SIGCOMM 2007*.

[56] Fonseca, R., Porter, G., Katz, R., Shenker, S., and Stoica, I. X-Trace: A Pervasive Network Tracing Framework. In *NSDI 2007*.

[57] Fonseca, R., Porter, G. M., Katz, R. H., Shenker, S., and Stoica, I. IP Options are not an option. Tech. Rep. UCB/EECS-2005-24, EECS Department, University of California, Berkeley, Dec 2005.

[58] Godefroid, P., Klarlund, N., and Sen, K. DART: directed automated random testing. In *PLDI 2005*.

[59] Gold, R., Gunningberg, P., and Tschudin, C. A Virtualized Link Layer with Support for Indirection. In *FDNA 2004*.

[60] Greenberg, A., Hjalmtysson, G., Maltz, D. A., Myers, A., Rexford, J., Xie, G., Yan, H., Zhan, J., and Zhang, H. A Clean Slate 4D Approach to Network Control and Management. In *ACM SIGCOMM Computer Communication Review. 35(5). October, 2005*.

[61] Greenberg, A., Lahiri, P., Maltz, D. A., Patel, P., and Sengupta, S. Towards a next generation data center architecture: scalability and commoditization. In *PRESTO 2008*.

[62] Gude, N., Koponen, T., Pettit, J., Pfaff, B., Casado, M., McKeown, N., and Shenker, S. NOX: towards an operating system for networks. *SIGCOMM Comput. Commun. Rev. 38*, 3 (2008).

[63] Gupta, P., and McKeown, N. Packet Classification on Multiple Fields. In *SIGCOMM 1999*.

[64] He, Y., Faloutsos, M., Krishnamurthy, S., and Huffaker, B. On routing asymmetry in the Internet. In *GLOBECOM 2005*.

[65] Huston, G. BGP Reports. Dec 2007, `http://bgp.potaroo.net`.

[66] Joseph, D., and Stoica, I. Modeling middleboxes. *Network, IEEE 22*, 5 (2008).

[67] Joseph, D., Tavakoli, A., and Stoica, I. "a policy-aware switching layer for data centers". In *SIGCOMM 2008*.

[68] Kandula, S., Chandra, R., and Katabi, D. What's Going On? Learning Communication Rules in Edge Networks. In *SIGCOMM 2008*.

[69] Karagiannis, T., Mortier, R., and Rowstron, A. Network Exception Handlers: Host-network Control in Enterprise Networks. In *SIGCOMM 2005*.

[70] Karsten, M., Keshav, S., Prasad, S., and Beg, M. An Axiomatic Basis for Communication. In *SIGCOMM 2007*.

[71] Katz, R. H., Porter, G., Shenker, S., Stoica, I., and Tsai, M. COPS: Quality of Service vs. Any Service at All. In *IWQoS 2005*.

[72] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click modular router. *ACM Transactions on Computer Systems 18*, 3 (August 2000), 263–297.

[73] Koponen,, Teemu and Chawla,, Mohit and Chun,, Byung-Gon and Ermolinskiy,, Andrey and Kim,, Kye Hyun and Shenker,, Scott and Stoica,, Ion. A Data-Oriented (and beyond) Network Architecture. In *SIGCOMM 2007*.

[74] Kopparapu, C. *Load Balancing Servers, Firewalls, and Caches*. Wiley, 2002.

[75] Lakshminarayanan, K. *Design of a Resilient and Customizable Routing Architecture*. PhD thesis, EECS Dept., University of California, Berkeley, 2007.

[76] Mahimkar, A., Dange, J., Shmatikov, V., Vin, H., and Zhang, Y. dFence: Transparent Network-based Denial of Service Mitigation. In *NSDI 2007*.

[77] Massey, D., Want, L., B.Zhang, and Zhang, L. A Proposal for Scalable Internet Routing and Addressing. In *Internet Draft draft-want-ietf-efit-00* (Feb 2007).

[78] Meyer, D., Zhang, L., and Fall, K. Report from the IAB Workshop on Routing and Addressing. In *Internet Draft draft-iab-raws-report-02.txt* (Apr 2007).

[79] Nalepa, G. J. *Advances in Intelligent Web Mastering*. ch. A Unified Firewall Model for Web Security.

[80] Narten, T. Routing and Addressing Problem Statement. In *Internet Draft draft-narten-radir-problem-statement-01.txt* (Oct 2007).

[81] Oppenheimer, D., Ganapathi, A., and Patterson, D. Why do internet services fail, and what can be done about it. In *USENIX Symposium on Internet Technologies and Systems* (2003).

[82] Pagiamtzis, K., and Sheikholeslami, A. Content-addressable memory (CAM) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits 41*, 3 (March 2006).

[83] Paxson, V. Bro: A system for detecting network intruders in real-time. *Computer Networks 31*, 23–24 (1999), 2435–2463.

[84] Reumann, J., and Shin, K. G. Stateful distributed interposition. *ACM Trans. Comput. Syst. 22*, 1 (2004).

[85] Roscoe, T., Hand, S., Isaacs, R., Mortier, R., and Jardetzky, P. Predicate routing: enabling controlled networking. *SIGCOMM Comput. Commun. Rev. 33*, 1 (2003).

[86] Singh, S., Baboescu, F., Varghese, G., and Wang, J. Packet Classification Using Multidimensional Cutting. In *SIGCOMM 2003*.

[87] Stevens, W. R. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, 1997.

[88] Stoica, I., Adkins, D., Zhuang, S., Shenker, S., and Surana, S. Internet Indirection Infrastructure. In *SIGCOMM 2002*.

[89] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM 2001*.

[90] Stoica, I., Shenker, S., and Zhang, H. Core-stateless fair queueing: a scalable architecture to approximate fair bandwidth allocations in high-speed networks. *IEEE/ACM Trans. Netw. 11*, 1 (2003).

[91] Taylor, D. E. Survey and taxonomy of packet classification techniques. *ACM Comput. Surv. 37*, 3 (2005), 238–275.

[92] Tennenhouse, D., Smith, J., Sincoskie, W., Wetherall, D., and Minden, G. A Survey of Active Network Research. *IEEE Comm. Magazine* (Jan 1997).

[93] Universal tun tap driver. `http://vtun.sourceforge.net/`.

[94] Walfish, M., Stribling, J., Krohn, M., Balakrishnan, H., Morris, R., and Shenker, S. Middleboxes No Longer Considered Harmful. In *OSDI 2004*.

[95] Wool, A. A Quantitative Study of Firewall Configuration Errors. *Computer 37*, 6 (2004).

[96] Yang, X., Wetherall, D., and Anderson, T. A DoS-limiting network architecture. In *SIGCOMM 2005*.

[97] Yoon, M. K., Chen, S., and Zhang, Z. Reducing the size of rule set in a firewall. In *Intnl. Conference on Communications, 2007*.

[98] Yu, F., Lakshman, T. V., Motoyama, M. A., and Katz, R. H. SSA: a power and memory efficient scheme to multi-match packet classification. In *ANCS 2005*.

[99] Zhang, Y., Breslau, L., Paxson, V., and Shenker, S. On the Characteristics and Origins of Internet Flow Rates. In *SIGCOMM 2002*.

[100] Zimmerman, H. OSI Reference Model - The ISO Model of Architecture for Open Systems Interconnection. *IEEE Trans. on Comm.* (April 1980).

Appendix A

PSwitch Internals

A.1 *Pswitch* frame processing

In this appendix, we provide a detailed description of how stateless and stateful *pswitches* process frames.

A.1.1 Stateless *Pswitch*

inP

The inP module associated with a *pswitch* interface X redirects incoming frames to appropriate classifiers based on policy. Non-IP frames like ARP are ignored by the inP module and pushed out to Switch Core interface X unmodified for regular Ethernet forwarding. In order to avoid forwarding loops, an inP module does not lookup policy and redirect frames that have already been redirected by another inP module. Such frames, identified by the presence of encapsulation, are emitted unmodified to Switch Core interface X.

Algorithm 2 lists the processing steps performed by the inP module when a frame f arrives at *pswitch* interface X. The following are the two main steps:

Step 1: Match rule:

The inP module looks up the rule matching f from the RuleTable. f is discarded if no matching rule is found.

Step 2: Determine next hop:

A successful rule match yields the classifier or server to which f is to be forwarded next. If the *Next Hop* of the matching rule specifies *FinalDestination*, then the server identified by f's destination MAC address is the next hop. If the *Next Hop* field lists multiple instances of a classifier, then the inP chooses a particular instance for the flow associated with f, by using flow direction agnostic consistent hashing on f's 5-tuple fields hinted by the policy (refer Section 5.3.3).

outP

The outP module of *pswitch* interface X receives the frames emitted by Switch Core interface X before they exit the *pswitch*. These frames will be in encapsulated form, having been processed by an inP module at the same or a different *pswitch* prior to entering the Switch Core. If *pswitch* interface X is connected to another *pswitch* and not to a server/classifier, then the outP module emits out the frame unmodified through *pswitch* interface X. If *pswitch* interface X is connected to a server/classifier, and the destination MAC address of the received encapsulated frame does not match the MAC address of the server/classifier, then the frame is dropped to avoid undesirable behavior from confused servers/classifiers. For example, a firewall may terminate a flow by sending a TCP RST if it receives an unexpected frame. If the destination MAC address of the encapsulated frame matches the MAC address of the connected server/classifier, then the frame is decapsulated and emitted out through *pswitch* interface X. The server/classifier receives a regular Ethernet II frame and appropriately processes it.

Algorithm 3 lists the processing steps performed by the outP module when it receives a frame f emitted by the Switch Core interface X.

Algorithm 2 INP processing in a stateless *pswitch*.

```
1: procedure INPPROCESS.STATELESS(interface X, frame f)
2:     if f is not an IP frame then
3:         Forward f to Switch Core interface X
4:         return
5:     end if
6:     if f is encapsulated then
7:         Forward f to Switch Core interface X
8:         return
9:     end if
10:    prvMbox = GetPrvHop(f)
11:    rule = RULETABLE.LookUp(prvMbox, f)
12:    if rule != nil then
13:        if rule.nxtHop != FinalDestination then
14:            nxtMboxInst = ChooseInst(rule.nxtHopInsts, f)
15:            encF = Encap(f, prvMbox.MAC, nxtMboxInst.MAC)
16:            Forward encF to Switch Core interface X
17:        else
18:            encF = Encap(f, prvMbox.MAC, f.dstMAC)
19:            Forward encFrame to Switch Core interface X
20:        end if
21:    else
22:        Drop f
23:    end if
24: end procedure
```

Algorithm 3 OUTP processing in a stateless *pswitch*.

```
1: procedure OUTPPROCESS.STATELESS(interface X, frame f)
2:     if interface X is not connected to a server/classifier then
3:         Emit f out of pswitch interface X
4:     else
5:         connMAC = MAC of connected server/classifier
6:         if f.dstMAC != connMAC then
7:             Drop f
8:         else
9:             decapF = Decapsulate(f)
10:            Emit frame decapF out of pswitch interface X
11:        end if
12:    end if
13: end procedure
```

An OUTP module can detect whether a classifier instance connected to it is dead or alive, using information from the FAILDETECT module. When emitting a frame to a dead classifier instance, the OUTP module has two options:

1. Drop the frame or,
2. Redirect the frame to a live instance of the same classifier type.

The first option of dropping frames destined to dead classifier instances keeps our design simple, and is an apt tradeoff when classifier failures are rare. The second option of redirecting frames to live classifier instances offers greater resiliency against packet drops. The *pswitch* which originally chose the failed classifier instance removes it from consideration in the classifier instance selection step when the news about failure eventually reaches it. Since the same selection algorithm is used at both the original *pswitch* and at the redirecting *pswitch*, the same classifier instance is chosen for the flow, hence reducing chances of flows that traverse stateful classifiers breaking.

Re-selection of classifier instances and redirection of frames by the OUTP module raise the specter of forwarding loops. For example, let firewall 1 be attached to *pswitch* 1 and

firewall 2 to *pswitch* 2. *Pswitch* 1 detects that firewall 1 has failed but *pswitch* 2 does not know about the failure yet and vice versa. *Pswitch* 1 redirects a frame destined to firewall 1 to firewall 2. When the frame reaches *pswitch* 2, it is redirected back to firewall 1. This creates a forwarding loop that persists till at least one of the *pswitches* hears about the failure of the firewall connected to the other *pswitch*. In order to prevent forwarding loops, each redirected frame includes a redirection TTL that limits the number of times a frame can be redirected by an outP module.

A.1.2 Stateful *Pswitch*

A stateful *pswitch* addresses some of the limitations of a stateless *pswitch* by storing per-flow state in the NextHopDb. The NextHopDb consists of two tables – FwdTable and RevTable. The two tables maintain per-flow state for the forward and reverse direction of flows, respectively.[1] Each table is a hash table with entries of the form *(*5-tuple, Previous hop MAC) → (Next hop MAC, TTL). Unlike classifier instance selection, the entire 5-tuple is always used in table lookup. Since a frame may traverse a *pswitch* multiple times during its journey, the previous hop MAC address is needed to uniquely identify entries. The TTL field is used to flush out old entries when the table fills up.

inP Processing

inP processing in the stateful *pswitch*, listed in Algorithm 4, is similar to that in a stateless *pswitch*. When the inP module receives an encapsulated IP frame, it looks up FwdTable for a next hop entry. This exact match-based lookup is faster than a pattern-based rule lookup. If a next hop entry is found, the frame is encapsulated in a frame destined to the MAC address specified in the entry and sent to the Switch Core. If a next hop entry is not found, a rule lookup is performed. If the rule lookup succeeds, the frame is encapsulated and forwarded to the appropriate server/classifier as in stateless inP processing. Additionally in stateful inP processing, an entry with the MAC address to which

[1] *Forward* is defined as the direction of the first packet of a flow.

Algorithm 4 INP processing in a stateful *pswitch*

```
1: procedure INPPROCESS.STATEFUL(interface X, frame f)
2:     Processing for non-unicast or non-data frames identical to Algorithm 2.
3:     prvMbox = GetPrvHop(f)
4:     nh = FWDTABLE.lookup(f.5tuple, prvMbox.MAC)
5:     if nh != nil then
6:         encF = Encap(f, prvMbox.MAC, nh.dstMAC)
7:         Forward encFrame to Switch Core interface X
8:     else
9:         rule = RULETABLE.LookUp(f)
10:        if rule != nil then
11:            if rule.nxtHop != FinalDestination then
12:                nxtMboxInst = ChooseInst(rule.nxtHopInsts,f)
13:                encF = Encap(f, prvMbox.MAC, nxtMboxInst.MAC)
14:                Forward encF to Switch Core interface X
15:                FWDTABLE.add([f.5tuple, prvMbox.MAC]→nxtMboxInst.MAC)
16:            else
17:                encF = Encap(f, prvMbox.MAC, f.dstMAC)
18:                Forward encFrame to Switch Core interface X
19:                FWDTABLE.add([f.5tuple, prvMbox.MAC] → f.dstMAC)
20:            end if
21:        else
22:            revnh = REVTABLE.lookup(f.5tuple, prvMbox.MAC)
23:            if revnh != nil then
24:                encF = Encap(f, prvMbox.MAC, revnh.dstMAC)
25:                Forward encFrame to Switch Core interface X
26:                FWDTABLE.add([f.5tuple, prvMbox.MAC] → revnh.dstMAC)
27:            else
28:                Error: drop f
29:            end if
30:        end if
31:    end if
32: end procedure
```

the encapsulated frame is forwarded is added to the FwdTable. If the rule lookup fails, RevTable is checked for a next hop entry associated with the flow, created by some prior frame of the flow in the opposite direction. If an entry is found, the frame is encapsulated and forwarded to the destination MAC address specified by the entry. For faster lookup on subsequent frames of the same flow, an entry is added to the FwdTable.

outP Processing

outP processing in the stateful Policy Core is identical to that in the stateless Policy Core except for the additional processing described here. As listed in Algorithm 5, while processing a frame destined to a directly attached classifier/server, a stateful outP module adds a next-hop entry to the RevTable. This entry records the last classifier instance traversed by the frame and hence determines the next classifier instance to be traversed by frames in the reverse flow direction arriving from the classifier/server. For example, the next-hop entry for a frame ($IP_A : Port_A \rightarrow IP_B : Port_B$) arriving from *firewall 1* destined to server B will be ($IP_B : Port_B \rightarrow IP_A : Port_A$, prevHop=*server B*, nextHop=*firewall 1*). The RevTable next-hop entry is used in inP processing if both FwdTable and policy lookup fail, and thus provides a default reverse path for the reverse flow direction.

The policy lookup in Step 9 of Algorithm 4 provides the flexibility to explicitly specify a different classifier sequence for the reverse flow direction. The RevTable lookup in Step 22 enables us to skip specifying the policy for the reverse flow direction. Per-flow state is used to automatically select the same classifier instances in reverse order. Thus, per-flow state simplifies policy specification. It also avoids expensive policy lookup and classifier instance selection operations on every frame by using the next hop classifier MAC address recorded in the FwdTable.

Algorithm 5 OUTP processing in a stateful *pswitch*

```
1: procedure OUTPPROCESS.STATEFUL(interface X, frame f)
2:     if interface X is not connected to a server/classifier then
3:         Emit f out of pswitch interface X
4:     else
5:         connMAC = MAC of connected server/classifier
6:         if f.dstMAC != connMAC then
7:             Drop f
8:         else
9:             decapF = Decapsulate(f)
10:            Emit frame decapF out of pswitch interface X
11:            prvMbox = GetPrvHop(f)
12:            rev5tuple = Reverse(f.5tuple)
13:            nh = FWDTABLE.lookup(rev5tuple, connMAC)
14:            if nh == nil then
15:                REVTABLE.add([rev5tuple, connMAC] → prvMbox.MAC)
16:            end if
17:        end if
18:    end if
19: end procedure
```