Packet Classification with Explicit Coordination

Mosharaf Chowdhury and Sameer Agarwal *
University of California, Berkeley

ABSTRACT

Packet classification is a key building block of many network services and functionalities (e.g., switching, filtering, load balancing). Despite its prevalence, packet classification is implemented and deployed in an ad-hoc manner at different layers of the protocol stack. Moreover, high speed packet classification, in presence of arbitrarily large number of classification rules, is resource and computation intensive.

We argue that instead of developing classification solutions in isolation, packet classification should be considered as a fundamental primitive in the protocol stack. We propose a classification layer or *CLayer* as part of the protocol stack and a generic mechanism to explicitly configure and implement capability-driven classification offloading. Our approach allows *classifiers* (e.g., routers, firewalls) to offload part of their classification related computation and memory requirements to *helpers* (e.g., end hosts, edge routers) without introducing additional states in the *classifiers*. Evaluation results from our prototype implementations attest *CLayer*'s scalability and show increasing performance gain as traditional classification complexity grows.

1. INTRODUCTION

Packet classification is a key building block of many crucial network services and functionalities across different layers. While traversing an end-to-end path in the current Internet, a packet encounters multiple network entities that perform packet classification – firewalls may classify the packet to decide whether to drop it or not, routers classify it to determine its next hop destination and QoS requirements, and a load balancer may classify the packet to direct it to a particular service instance. Packet classification is only going to become even more pervasive in the future. As networks evolve toward using commodity hardware (e.g., OpenFlow [7]) for greater flexibility and simpler management, classification is becoming an integral part of the architecture.

High speed packet classification, in presence of arbitrarily large number of classification rules, is known to be computation-intensive, power hungry, and consequently, expensive [15]. It leads to high implementation and configuration complexity as well as difficulties in accurately capturing classification semantics.

Classification involving multiple fields exhibits a fundamental trade-off between computational complexity and memory requirements [15]. Routers and firewalls commonly utilize expensive and power-hungry TCAMs for fast pattern-based classification on multiple packet header fields for every single packet. Classification involving packet payloads is even more complex.

*This is a joint work with Dilip Joseph and Ion Stoica.

Table 1: Packet classification across different layers of the network protocol stack

Layer	Network service/functionality
Link (2.5)	Switching, MPLS
Network	Forwarding
Transport	Filtering, IntServ, DiffServ
Application	Load balancing, Intrusion detection

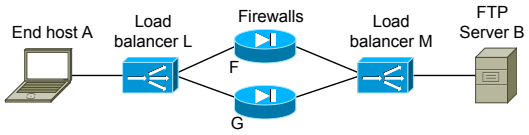

Figure 1: Coordinating multiple *classifiers* for firewall load balancing.

The lack of support for explicit coordination between different entities involved in packet classification makes configuration hard. Consider the pair of firewall load balancers in Figure 1. The load balancers must select the same firewall instance for both forward and reverse directions of a TCP flow, since the classification decision at a firewall instance uses stored information about packets previously seen in the reverse direction. To achieve this level of coordination, today's load balancers have to use ad-hoc mechanisms that leverage the physical wiring configuration and the 5-tuples of the packets received on each network interface [20].

Typically, there is a semantic gap between the end-points and the entities performing classification. For example, an edge router is in a better position to determine a packet's QoS. Its closeness to traffic sources provides access to finer-grained local QoS policies and traffic accounting than a core router.

Despite its prevalence, packet classification is implemented and deployed in an ad-hoc manner at different layers of the protocol stack. Most implementations follow roughly the same sequence of actions – a *classifier* determines which flow or class an arrived packet belongs to, looks up rules for that class, and takes actions based on corresponding rules. Existing approaches to ameliorate the aforementioned challenges are also limited to reinventing the wheel. For instance, to improve classification performance and to reduce semantic gaps, several solutions (e.g., MPLS, DiffServ) have proposed

pushing packet classification tasks to edge network elements that handle less traffic and have more semantic context. The lack of architectural support for a general mechanism forced each of these solutions to devise their own set of protocols.

We argue that rather than developing ad-hoc classification solutions in isolation, packet classification should be included as a fundamental primitive in the protocol stack. Furthermore, it should be simultaneously accessible from different applications and services to avoid unnecessary reimplementation of common functionalities. In this report, we propose a *classification layer* or *CLayer* as part of the protocol stack and a generic mechanism to efficiently configure and implement capability-driven classification offloading using *CLayer*. We introduce the concept of *Fate-Carrying Labels* (*FCLs*) that allow *classifiers* – entities like routers, firewalls, and load balancers that traditionally perform packet classification – to include *helpers* – end hosts, edge routers, and other network entities that have more semantic context – in classification-related computation and memory requirements in a per-flow basis without introducing any additional states in the classifiers.

In developing *CLayer*, we have (i) designed a generic solution that can simultaneously handle a wide variety of existing and future classification applications and (ii) a robust signaling protocol that can handle non-symmetric paths, path changes, and state discrepancies at participating entities. (iii) We have made offloading robust to tampering, cheating, or malfunctioning of *helpers*, and (iv) developed an API interface to easily *CLayer*-enable existing applications.

We have prototyped *CLayer* in software using the Click [19] modular router and have provided *CLayer* support for a variety of existing network applications with minor modifications to their source code. Our experience with *CLayer*-enabled applications provide several insights into the benefits of using *CLayer*: Our implementation of a *CLayer*-enabled firewall can achieve two to three times more throughput than a regular firewall implementation over multiple rule sets, and we have observed noticeable performance improvement after enabling *CLayer* in layer-4 load balancers. *CLayer* also reduces network complexity by obviating separate implementation and configuration of diverse mechanisms.

The remainder of the report is organized as follows. Section 8 puts *CLayer* into perspective with its related work. Section 2 and Section 3 explain the basics of *CLayer*. We discuss additional design and performance issues of *CLayer* in Section 4. Section 5 and Section 6 cover *CLayer* prototyping details followed by evaluation results. We outline currently known limitations of *CLayer* in Section 7, then put forth our future work plan in Section **??**, and conclude with a summary in Section 9.

2. OVERVIEW

In this section we present an overview of *CLayer*. We start with the basic classification model and signaling mechanism, then briefly describe properties and format of *labels* in *CLayer*, and finally make our case for a new classification layer.

2.1 Classification Model

Network entities can be assigned two different roles in *CLayer*: *classifiers* and *helpers*. *Classifiers* are network entities like routers, firewalls, load balancers, and other middleboxes that traditionally perform packet classification. *Helpers* aid *classifiers* by performing classification tasks on their behalf. Examples of *helpers* include end hosts that provide HTTP cookies to web load balancers to aid them in session identification and edge routers in a DiffServ domain that set code points in packet headers to be used by core routers in packet processing.

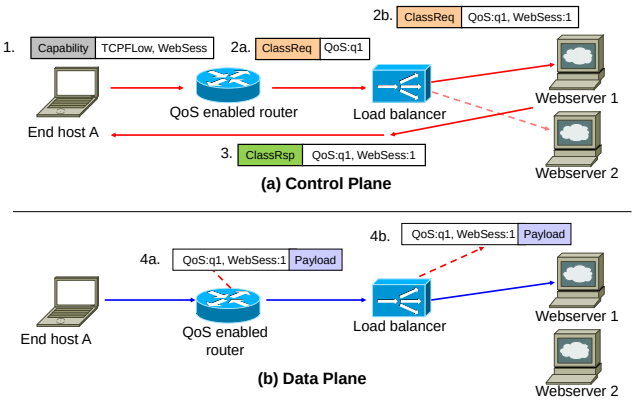

Figure 2: Packet classification using *CLayer*.

The overall *CLayer* proposal consists of three main components:

1. A signaling protocol that coordinates *classifiers* and *helpers*.
2. The *CLayer* header that carries signaling information and classification results using *Fate-Carrying Labels* (*FCLs*).
3. A new socket API that provides *CLayer*-interaction interface to network services and applications.

The purpose of *CLayer* is to provide a generic classification mechanism and at the same time offloading part of computation and memory requirements of *classifiers* to relevant *helpers* in order to improve performance and scalability. The *CLayer* classification model is simple (as illustrated in Figure 2): using a robust signaling protocol, *helpers* advertise their classification and labeling capabilities (**1**). Based on the first few packets, a *classifier* classifies the flow (**2a** and **2b**) and requests the originator *helper* about the type of classification expected from it, i.e., what *label* the *classifier* expects to find in subsequent packets (**3**). The *helper* maintains *label* information using *soft*-states, and it embeds the requested *label* in later packets. The downstream *classifier* simply reads and uses these *labels* to speed up its classification operations (**4a** and **4b**).

2.2 Basic CLayer Signaling Protocol

The *CLayer* signaling protocol involves a four-way handshake – *CL_SYN* - *CL_SYNACK* - *CL_ACK1* - *CL_ACK2* – appropriately named to highlight its similarity to the three-way *SYN-SYNACK-ACK* handshaking protocol of TCP. *CLayer* messages can carry different types of information inserted by entities on the path between the two end points (Table 2). When and why these information are generated and who uses them will become clear as we explain in subsequent sections.

Table 2: Summary of different types of information carried in *CLayer* messages

Information	Related to...
Capability	Capability declaration by a *helper*.
ClassReq	Classification request from a *classifier*.
EchoReq	Classification request echoed once by an end point *helper* toward the intended *helper*.
InstallReq	Echoed request reechoed by the opposite end point *helper*. This can happen when the intended *helper* is not an end point.
Result	*Label* requested by a *classifier*.

We illustrate the basic functionality of *CLayer* using the example in Figure 3, where end host A wishes to communicate with end host B and router E on the data path classifies packets based on its QoS policy and assigns different forwarding priorities. E is thus the *classifier* and uses the *CLayer* signaling protocol to configure *helpers* A and B. Here, *CLayer* provides benefits similar to DiffServ and CSFQ [23] – router E does not have to perform expensive packet classification on every packet, nor does it have to maintain per-flow state.

When A initiates communication with B, it first sends a *CL_SYN* to B. *Helpers* on the $A \rightarrow B$ data-path advertise their capabilities and *classifiers* place classification requests (*ClassReqs*) through the *CL_SYN*. 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 *labels* 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 3 illustrates the *CLayer* signaling messages exchanged between A and B, 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.

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 label q_1 denoting the assigned QoS class.

Step 3: B responds to the *CL_SYN* message 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 include the classification results of their respective *ClassReqs* in every subsequent data packet they exchange. It is assumed that after reading information addressed to itself, each entity invalidates that information from the outgoing packet.

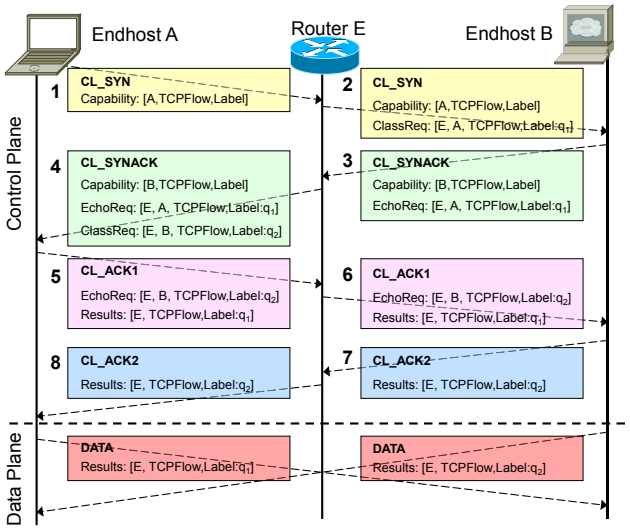

Figure 3: Detailed *CLayer* signaling in a QoS application.

In this scenario, *CL_ACK2* is unnecessary; a three-way handshake suffices. However, as we describe in detail in Section 3.1, *CLayer* signaling includes a 4-way handshake to handle non-end host *helpers* in the presence of network path asymmetry.

The signaling illustration here is described as a standalone protocol for simplicity of explanation. In practice, it is piggybacked on top of TCP handshake and initial data packets of a connection, thus avoiding extra round trip overheads.

2.3 Fate-Carrying Labels

A *label* in *CLayer* is an opaque bag of bits that is issued by a *classifier* and can be meaningfully interpreted only by its issuer. In its simplest form, a *label* can be a pointer in a $\langle label \rightarrow action \rangle$ lookup table. After receiving and verifying a previously issued label, a *classifier* will lookup its corresponding action to proceed. Since the number of possible actions is generally much lower than the number of rules that result in those actions, even in this straightforward interpretation *labels* can improve performance with a small state requirement for lookup tables.

In order to completely remove the state requirement, we propose *Fate-Carrying Labels* (*FCLs*) where a *Label* is not just a pointer to find out the corresponding action from a lookup table; rather it is the action itself. For example, packets from an already classified-to-be-dropped flow might come to a firewall with an *FCL* that says *"Drop Me"*! Consequently, there is no extra state in *classifiers* and no intermediate per-packet lookup stage using *FCLs*.

2.3.1 Properties

An *FCL* is expected to have the following properties for classification decision enforcement, incentives, trust, and security purposes.

- *Verifiable:* A *classifier* should be able to verify a *label* it issued in a fast and efficient manner. *Labels* should be *unforgeable*, or at least they should be very hard to forge or to randomly guess by anyone other than the issuing *classifier*. *Classifiers* should also be able to differentiate between malformed and corrupt labels. We consider a *label* to be 'malformed' or 'wrong' if it is unsuccessfully tampered with; otherwise, it is considered to be 'corrupt'.

- *Bound to flow:* A *label* should only be valid for the flow it was issued for. It also implies that *labels* should be *non-transferable* to another flow, and they are for *single-use* only – even for the same *helper*.

- *Limited valid time-window:* A *Label* corresponding to an action should be periodically invalidated to make it harder for anyone to learn them over time. Periodic invalidation also provides a natural way to throttle/deny misbehaving *helpers*.

Note that these properties are not correctness requirements and preserving them remain up to the discretion of an issuing *classifier*. Details of the importance of these properties are further explained in later sections.

2.3.2 FCL Format

In its stripped-down form, an *FCL* consists of a bit string `ActionRep` that represents the action the issuing *classifier* must take upon receiving this *FCL*. In order to detect a corrupted *label* and to assist in *label* invalidation, an *FCL* should also contain a `CheckSum` and an issuing `TimeStamp`.

Leaving such information in plain-text, makes them unverifiable and allows malicious entities to forge them. This is specially critical for *classifiers* like firewalls. To prevent that, an *FCL* can also include an HMAC (e.g., MD5, SHA-1) – on `ActionRep` and `TimeStamp` along with the 5-tuple of the flow to bind it – keyed using a `Secret` known only to the issuer. Upon receiving a packet, a *classifier* can now quickly authenticate and act upon it with a high-degree of confidence.

2.4 A Layer for CLayer

We propose a new *classification layer* in the network protocol stack to carry the *CLayer* signaling messages and the classification results provided by *helpers*. In order to simultaneously support classification applications at different layers and those that span multiple layers, *CLayer* logically spans layers 2 to 7 (link layer to application layer).

Although *CLayer* logically spans multiple layers, we advocate implementing it as a new layer between the network and transport layers to minimize violation of the layering principle. *CLayer* must be at the network layer or above in order to avoid extensive changes to current forwarding infrastructures. However, implementing the *CLayer* at or above the network layer still violates architectural layering for layer-2 only applications.

Placing *CLayer* at or above the network layer ensures that it is preserved end-to-end in the absence of layer-7 'proxies' on the path. A layer-7 proxy (e.g., 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 at least enough to blindly forward along any *CLayer* headers they receive.

A new layer naturally calls for its own header. We will defer header-related discussion to Section 5.

3. USAGE SCENARIOS

In this section we present two use cases in details that illustrate how *CLayer* can simplify network configuration complexities involving multiple *classifiers* by leveraging application semantic and by promoting explicit coordination between different entities. We also clarify why *CLayer* needs a four-way handshaking protocol to support non-end host *helpers*.

3.1 Multiple Classifiers and Leveraging Helper Context

We illustrate how *CLayer* can simultaneously support different classification operations on the data path – IP forwarding, QoS, and load balancing – using the following use case. It also demonstrates how application semantic available at a *helper* can significantly simplify a *classifier* (i.e., load balancer) operation.

Figure 4 illustrates a web browser running on host A sending HTTP requests to a web server (*httpd*) located in a data center. Load balancer L spreads out HTTP requests from end hosts across the web servers in the data center. Similar to the HTTP cookie mechanism, L leverages the semantic context available at A to send all its HTTP requests to the same web server W_1. Edge router E performs priority based forwarding with help from A and W_1 (see Section 2.2). Similar to MPLS, core router C offloads IP route lookup to edge routers E and F to reduce processing and memory requirements of C. In this example, E, C and L are *classifiers* and A, W_1, E and F are *helpers*.

Before sending the first HTTP request to L (whose address is IP_L), the browser at A creates a new *CLayer* session. Subsequently, the browser embeds the session handle into all TCP connections associated the HTTP session. Figure 4 shows how various *classifiers* and *helpers* participate in the *CLayer* signaling protocol. This protocol is piggybacked on the session's first TCP connection and consists of four messages.

Message 1: The browser at A sends a *CL_SYN* message to advertise its *HTTPSess* and *TCPFlow* classification capabilities (*CL_SYN* is piggybacked on TCP's *SYN* packet). Upon receiving *CL_SYN*, E advertises its ability to perform destination based packet classification by appending a *ClassReq* request. Next, C looks up the route for *CL_SYN* and appends a *ClassReq* request to ask E to label all packets it sends to IP_L with x. F also advertises its ability to perform destination based classification (like E), but does not insert a *ClassReq* request as it is not QoS-aware. Finally, L selects a web server W_1 based on current load conditions. Before forwarding the *CL_SYN* message to W_1, L appends a *ClassReq* request in which it asks A to label all packets in the same HTTP session with W_1.

Message 2: Upon receiving the *CL_SYN* message, W_1 replies with a *CL_SYNACK* message advertising its capability list and echoing the three *ClassReqs* from *CL_SYN*. L forwards the *CL_SYNACK* message unmodified, as it does not classify packets destined to end hosts. Next, F appends its capability advertisement, and C appends a *ClassReq* message, asking F to label all packets it sends to IP_A with y. Finally, E appends a *ClassReq* message asking W_1 to label all packets in the same TCP flow with q_2.

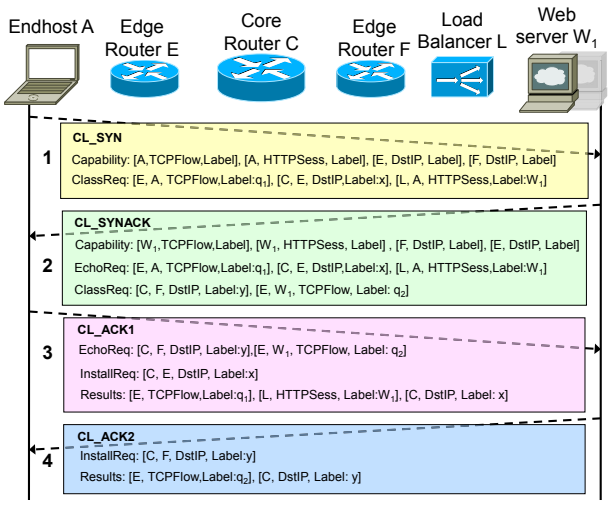

Figure 4: Using *CLayer* to offload QoS labeling, IP route lookup, and HTTP session identification to end hosts and edge routers.

Message 3: Upon receiving the *CL_SYNACK* message, A records all *EchoReqs* addressed to it and replies with a *CL_ACK1* message. In this message, A copies all the other *EchoReqs* as *InstallReqs*, as well as the classification results (*label*:q_1 and *label*:W_1). Next, E records the *InstallReq* addressed to it and appends the specified classification result (*label*:x). C uses label x to quickly identify the outgoing interface for *CL_ACK1* without performing classification, while F forwards the *CL_ACK1* unmodified. Finally, L uses label W_1 to select and forward the packet. Note that L does not need to reconstruct the TCP stream or parse the HTTP header.

Message 4: Upon receiving *CL_ACK1*, W_1 records the *EchoReqs* addressed by E to itself, and replies with a *CL_ACK2* message. This message contains the *EchoReq* addressed to F (copied in *InstallReqs*), and the classification result *label*:q_2. Next, L relays the *CL_ACK2* unmodified, while F records the *InstallReq* addressed to it and includes the requested classification result *label*:y. C uses this label to quickly forward the packet, while E uses label q_2 to forward the packet to A using the appropriate QoS. This completes the *CLayer* signaling protocol.

The possibility of asymmetric network paths (e.g., due to Internet path diversity [16] or load balancing Direct Server Return mode [20]) creates the need for the *CL_ACK2* message and makes *CLayer* signaling four-way instead of three-way. A non-end host *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*.

3.2 Explicit Coordination between Helpers

We revisit the scenario in Figure 1 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 to establish common state at different *helpers*.

Suppose end host 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 [20], 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 of an FTP session. Thus, current firewall load balancing solutions are complex both in terms of device implementation as well as in configuration.

CLayer 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 end hosts, just like web load balancers offloaded HTTP session identification to end hosts in Section 3.1.

L adds two *ClassReqs* to A's *CL_SYN* – (i) *ClassReq* c that directs end host 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 *CLayer*-enabled firewall load balancer thus simply reads the *label* in a packet's *CLayer* header and forwards it to the firewall instance denoted by that *label*. Such operational simplicity makes it feasible to integrate firewall load balancing functionality into routers and switches, avoiding the need for expensive special-purpose firewall load balancers.

4. ADDITIONAL DESIGN AND PERFORMANCE ISSUES

The *CLayer* design is intended to be (among other properties) self-incentivizing, robust, efficient, scalable, secure, incrementally deployable, and compatible with legacy applications after minimal changes. In this section we discuss these issues and some details of the design that are relevant to them.

4.1 Incentives for Using CLayer

While *classifiers* are intuitively incentivized by the new functionality and the possibility of complexity reduction enabled by *CLayer*, it may seem that *helpers* will not be interested in putting *labels* in the first place. We believe that the following inherent and imposed incentives will be able to convince them in doing so.

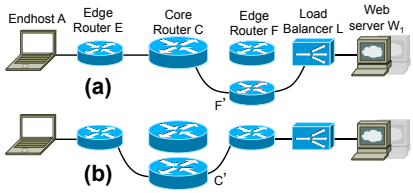

Figure 5: Path changes.

First of all, similar to incentives for participation in TCP congestion control, *helpers* are incentivized to help *classifiers* on their packets' path (even those in different administrative domains) for visibly better performance (i.e., higher throughput, lower latency).

Secondly, *CLayer*-enabled *classifiers* can adopt a policy to prioritize labeled packets over normal ones during congestion periods. Consequently, *helpers* will have to enable *CLayer* support in ordered to fairly compete for congested resources.

Finally, a *helper* might seem to be less incentivized to participate if it is insensitive to lower performance or when it has a conflict of interest with a *classifier* (e.g., a remote firewall that wishes to drop its packets). We believe that the periodic label invalidation policy will make it harder for a *helper* to learn about conflicting interesting without investing significant amount of resources. Even if it can learn and decide not to label packets, in the worst case a *CLayer*-enabled *classifier* will have to resort to per-packet classification which is no worse than the best case of a non-*CLayer classifier*.

4.2 Robustness

CLayer signaling is robust to path changes, lost or retransmitted messages, unexpected state expirations in *helpers*, and in handling malformed/corrupt/missing *labels*.

4.2.1 Path Changes

Changes that do not alter the sequence of *classifiers* and *helpers* on a packet's path have no impact on *CLayer* signaling. For example, common local layer-2 and wide area Internet path changes do not affect *CLayer* 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 *labels* and can be fixed by *re-signaling*. Figure 5 illustrates path changes in the example topology described in Section 3.1. To summarize, C and L are *classifiers*; A, W_1 and F are *helpers*; E is simultaneously a *helper* and a *classifier*. In Figure 5(a), the network path between end host A and web server W_1 shifts from edge router F to F'. In Figure 5(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, *helpers* re-run the *CLayer* 4-way handshake. The session handle established during original signaling is included in the *CLayer* headers. The *helpers* on the original path use the handle to retrieve the previously established states and insert *labels* in the re-signaling messages. *Classifiers* append *ClassReqs* only if they do not find the desired classification results addressed to them.

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 web server based on the *label*:W_1 embedded by *helper* A, irrespective of the path changes in Figure 5. 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 regular route lookup, although more expensively than using the label embedded by F.

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

4.2.2 Unexpected State Expiration

CLayer handles unexpected state expiry at *helpers* and *classifiers* by re-signaling. For example, in Figure 5, 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 end host 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 2.2 and 3.1). However, in some scenarios, they are related. For example in Section 3.2, end hosts A and B use the same label, so that the firewall load balancer pair can select the same firewall instance in both flow directions. If the state at B expired before A and A sends a packet to B, B will not be able to include the correct classification results in its response packet to A. The *CLayer* at B detects the absence of state identified by the session handle in the packet, and runs an out-of-band *CLayer* signaling to re-establish the missing state before replying to A. Out-of-band signaling can be avoided if A preemptively includes the shared *ClassReqs* in the data packet if it suspects that B may have forgotten the state – e.g., when sending a new packet after a long gap or when starting a new TCP connection.

We assume that the session handle (see Section 5) 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 end host C will not propose h_A and B will not reselect h_B before the $h_A.h_B$ state at A has expired.

4.2.3 Retransmitted Messages

CLayer 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 3.1, load balancer L selected the web server instance W_1 on processing the *CL_SYN* from A. Suppose A retransmits *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 latest *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 can send a TCP RST with the appropriate classification label embedded in the *CLayer* header.

4.2.4 Malformed or Corrupt Labels and Missing Labels or Headers

A *CLayer*-enabled *classifier* can differentiate between malformed and corrupt *labels* using per-*label* checksums and *label* verification mechanisms. If a *classifier* finds a malformed *label*, it just drops the packet. Otherwise, if the *label* is corrupt, the *classifier* considers this to be an exception; instead of dropping the packet, a per-packet classification is performed in this case. Packets with missing *labels* can trigger re-signaling, and packets without *CLayer* headers are given lower priority in times of congestion.

Note that per-packet classification is not always an option. Depending on specific application/service scenario, there can be different repercussions. For example, since there are no flow-specific states in *classifiers*, lost *labels* can result in connection drops in a load-balancer. For a firewall, on the other hand, a missing *label* might just require one more classification. But in both cases, *CLayer* performs no worse than the existing solutions.

4.3 Scalability

CLayer is scalable both with respect to signaling overhead and memory/state requirements in *helpers*. *CLayer* signaling is performed only at session start and when explicitly initiated after significant path change or state loss. Moreover, it does not introduce additional round-trips as it is piggybacked on packets of existing connection oriented protocols.

CLayer requires per session states only in *helpers*. Such state requirements do not restrict *CLayer* scalability as *helpers* like end hosts and less-loaded edge routers are typically not bottlenecked by memory. Moreover, *CLayer* memory requirements are small. For instance, at an end host less than ten bytes are needed per *CLayer* session, each consisting of one or more TCP connections.

Classifiers simply use classification results embedded by *helpers*. The decision of directly using actions – instead of introducing another level of $\langle label \rightarrow action \rangle$ indirection – obviates any additional state requirements in *classifiers*. In some cases, *CLayer* even reduces the state at *classifiers*. For example, a *CLayer*-enabled load balancer need not maintain $\langle flow \rightarrow server \rangle$ instance mappings.

4.4 Trust

CLayer does not require *helpers* or *classifiers* to trust each other. *Helpers* can ignore any classification requests from any *classifier*. But that can result in lower performance, because packets without *labels* are subject to regular rule matching and will be the first ones to be dropped when the *classifier* is overloaded.

Classifiers can avoid trusting helpers by using verifiable *FCLs*. As described in Section 2.3, for critical services a *classifier* can resort to authentication mechanisms to make sure that the *helper* has indeed put the same *FCL* that it is supposed to.

Unlike active networking [24], neither *helpers* nor *classifiers* execute code supplied by non-trusted entities. This further lowers overall trust requirements.

4.5 Security

We emphasize that our main goal here is not to design a bulletproof system. Instead, we aim to design simple and efficient solutions that make *CLayer* not worse and in many cases better than todaySs Internet. The solutions outlined here hence should only be viewed as a starting point toward more sophisticated and better security solutions.

4.5.1 Eavesdropping and Label Spoofing

A malicious attacker can try reusing *labels* assigned to another *helper*. If security-enabled *FCLs* are used, this attack is meaningful only in a pathological scenario: the malicious entity can bypass a *classifier* like firewall by spoofing the *label* and the 5-tuple of the corresponding flow, if it is in the same LAN and trying to communicate to the same end point as the *helper* (because *FCLs* are bound to the *helper* and the other end and if the attacker is not in the same LAN it will not be able to make use of it). Otherwise, *classifiers* can use the verifiability of *FCLs* to detect any tampering only to drop such packets.

4.5.2 DoS Attacks

We consider two types of DoS attacks: (a) attacks on *classifiers*, and (b) attacks on *helpers*. The fact that *CLayer* does not introduce any additional states in *classifiers* makes them resilient to DoS or DDoS attacks. The best an adversary can do is to send packets without any *labels*, which might make a *classifier* to fall back to per-packet classification. However, this is no worse than existing solutions, and if there are packets with *labels* a *CLayer*-enabled *classifier* will prioritize them over non-labeled ones anyway - thus foiling DoS attempts.

As for *helpers*, *CLayer* has small state requirements. However, as already described, *helpers* can always refuse to honor any classification request. A *helper* can set a threshold for the maximum allowable *CLayer*-related resources, and if it detects a *ClassReq* flooding, the *helper* can always deny them after the threshold is crossed.

4.5.3 Malicious Label Modification

An attacker can try to maliciously change a *label* to adversely affect the outcome. Since *FCLs* are verifiable, *classifiers* will be able to detect such modifications and just drop such packets. One can imagine one more pathological case where an attacker corrupts a *label* to get it dropped by *classifiers*. However, if the sole intention is to drop a packet and the attacker has access to packets to corrupt them in the first place, it can just drop them by itself instead of going

through all the trouble.

4.6 Privacy

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

Our *CLayer*-based firewall design overcomes rule secrecy restrictions by relying on first packet classification locally at the firewall itself. End hosts simply reflect the *FCL* supplied by the firewall, unaware of the filtering rules that derived the *label*. Moreover, periodic invalidation makes it almost impossible to learn which action a *label* represents. An additional benefit of our 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 end hosts, 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 end hosts by default.

4.7 Deployment Issues

4.7.1 In the Internet

An Internet-wide *CLayer* deployment does not require a fork-lift upgrade of the entire network. *CLayer* traffic can coexist with non-*CLayer* traffic. Only entities wishing to benefit from *CLayer* need to be upgraded. However, a *CLayer*-enabled connection between two end hosts requires all layer 4 and above *classifiers* and forwarding elements on the path between them to be *CLayer*-aware or at least to ignore *CLayer* headers and forward packets unmodified. A non-*CLayer*-aware layer-4 router or firewall may consider *CLayer* packets as malformed/suspicious and thereby drop them, irrespective of whether the *CLayer* header is implemented as a separate layer or is tucked into a new IP Option. Hence, an end host must fall back to a non-*CLayer*-enabled connection if *CLayer* signaling to a new destination does not successfully complete even after multiple attempts.

4.7.2 Split Proxy-based Solution

CLayer requires both end points of a connection to be *CLayer*-aware for proper functioning, which can be a major hindrance toward its wider acceptance. One way to address this problem is to introduce *CLayer*-enabled *split* proxies in edge networks instead of changing the network protocol stacks in all the end hosts. In this case, the proxy will take care of *CLayer*-headers by adding and removing them on outgoing and incoming paths respectively. Such a proxy can be inserted into a network as any other middlebox. However, state requirements in *helpers* will move to the split proxy, and its failure will result in connection disruptions in all the concerned *helpers*.

4.7.3 In Data Center and Enterprise Networks

A data center or an enterprise network is an easier candidate for *CLayer* deployment than the wide area Internet. The single administrative domain can enable easier modification to end hosts as advocated by other new network architectures. Moreover, multiple new data centers being built today offer hope for a clean-slate *CLayer* implementation. A *proxy* at the ingress can cleanly separate the data center network from the Internet and add appropriate *CLayer* headers. Implementing such a proxy that can scale to data center workloads is an open challenge. This approach however confines *CLayer* functionality and benefits to within the data center.

4.8 Legacy Applications

CLayer deployability also depends on its ease of implementation and integration into existing networks. Even though *CLayer* requires modifications to *helpers* and *classifiers*, its close similarity to BSD socket libraries and the small number of lines to port existing applications demonstrate that *CLayer* can be easily integrated into existing applications (see Section 5). *CLayer* daemon functionality can be embedded in future OS versions or can be installed as a standalone system service.

4.9 Inter-domain Concerns

CLayer does not introduce any additional concerns in terms of security, privacy, and trust between different administrative domains. *Classifiers* in one domain need not give away any information to *classifiers* in another.

As already discussed, an *FCL* is a collection of random meaningless bits to everyone else other than the issuing *classifier*. Moreover, they can be periodically changed, and a *classifier* can introduce additional obfuscation methods.

If a *classifier* in one domain selfishly overwrite or remove *labels* provided by *classifiers* in other domains, some or all of *CLayer*-enabled *classifiers* might fail to observe the expected performance gains. Even in such pathological cases, *CLayer* can fall back to per-packet classification, which is as good as the existing solutions.

4.10 Supporting Connectionless Protocols

Even though we have described and evaluated *CLayer* by piggybacking its handshaking messages on TCP, it can also be implemented on top of UDP-like connectionless protocols by introducing some modifications to its semantics. In this case, *CL_SYN* - *CL_SYNACK* - *CL_ACK1* - *CL_ACK2* – the whole four-way handshaking protocol becomes superfluous. Instead of piggybacking, *classifiers* just need to generate additional packets to propagate *ClassReqs* back to the corresponding *helpers* whenever they see *CL_SYN* messages with *helper* capabilities. *EchoReqs* and *InstallReqs* will also become unnecessary.

However, additional packet generation requires CPU and memory resources. Thus it can attract DoS or DDoS attacks on *classifiers* similar to TCP/IP *SYN* flooding.

5. IMPLEMENTATION DETAILS

CLayer adds special packet headers, and *CLayer*- enabled applications and services require *CLayer*-specific API to utilize it. This section describes both parts and gives an overview of our prototype implementation.

5.1 CLayer Header

Ideally, *CLayer* header has a free-form, flexible length, key-value format. But any format that provides required performance at relevant network entities can be used.

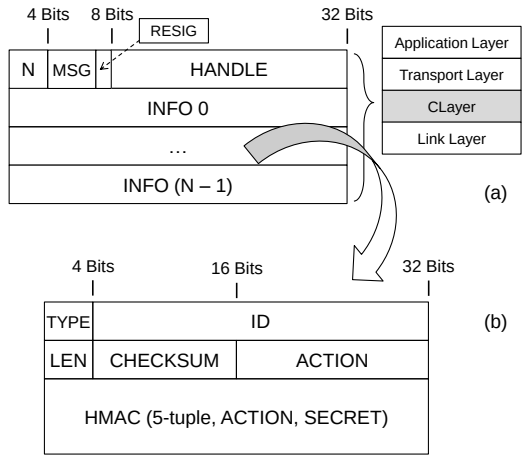

Figure 6: (a) A rigid *CLayer* header format and *CLayer* location in the network protocol stack; (b) An example *Fate-Carrying Label*.

Figure 6(a) shows a rigid header format optimized for forwarding performance of entities like routers which work faster on simple header formats. The 4-bit field *N* specifies the number of information pieces (0 to 15) that follow. Recall that there can be 5 types of information (Table 2), 4 are exclusively for handshake and the last for carrying actual *labels*. The 3-bit *MSG* field refers to one of the 4 handshaking messages or *DATA* otherwise. The 1-bit *RESIG* field is set only during the resignaling phase (see Section 4.2). The 24-bit *HANDLE* uniquely identifies the *CLayer* session associated with the results.

The *HANDLE* is a concatenation of bits randomly proposed by the two end hosts in the *CL_SYN* and *CL_SYNACK* messages. All *CLayer*-related state at *helpers* and *classifiers* is keyed by this handle. In Section 4.2, we described how this handle plays an important role in making *CLayer* signaling robust.

Each *INFO* consists of (i) a 4-bit *TYPE* field denoting what type of information is this; (ii) a 4-byte *ID* field denoting the entity to which this information piece is addressed; (iii) a 4-bit *LEN* field specifying the length of the total *INFO* in multiples of 4 bytes; (iv) a 12-bit *CHECKSUM* field containing the checksum of the complete *INFO*; The rest are dependent on the *helper* or *classifier* that issued this *INFO*. Figure 6(b) represents an *FCL* which consists of an 16-bit *ACTION* field followed by a *HMAC* of all the information that must verifiable at the *classifier*.

Since there are often multiple *classifiers* and *helpers* on a packet's path, each *INFO* must be explicitly addressed. These ids need not be globally routable. They just need to be unique on the path of a particular data flow. We use existing identifiers (in this case part of IP addresses) to identify *helpers* and *classifiers*.

5.2 CLayer API

Applications at end hosts interact with *CLayer* using the *CLayer* network library. We suggest a socket library very similar to the standard BSD socket library so that existing network applications can be easily ported. However, *CLayer* is not restricted to this particular API.

Our library consists of functions like `cl_connect` and `cl_bind` that have direct semantic correspondence with the standard BSD socket library and *CLayer*-specific functions like `cl_session_create` and `cl_add_capability`. A *CLayer*-aware application has a structure very similar to that of any common application using the BSD library. `cl_add_capability` enables an application to advertise its semantic classification capabilities (e.g., label all packets in an HTTP session). `cl_session_create` creates new session state of the specified type (e.g., TCP_FLOW, FTP_SESS, WEB_SESS) and returns a handle to the application. The application associates a TCP connection with a session by passing the session handle to `cl_connect`.

The application sends and receives data using standard `send` and `recv` socket calls. *CLayer* processing module in the OS performs the classification tasks configured during *CLayer* 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.

Table 3: SLOC of our prototype implementation

Component	SLOC
lighttpd web server	19
httperf HTTP benchmark tool	7
wget command line HTTP client	10
Layer-4 firewall	308
Layer-4 load balancer	190
CLayer socket library & daemon	4025

5.3 Prototype Implementation

We prototyped *CLayer* using Click [19] software modular router and *CLayer*-enabled a variety of *helper* and *classifier* applications and network services for evaluation purposes.

CLayer implementation in a *helper* node consists of two parts – a daemon and a network socket library. The daemon implements the core *CLayer* functionality, i.e., control plane signaling and data plane classification. *Helper* applications interact with the daemon through *CLayer* socket library calls, as described in Section 5.2. 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* device to intercept outgoing packets and to transfer incoming packets to regular network processing after stripping out *CLayer* headers.

CLayer-enabling an existing *helper* application often simply involves replacing BSD socket calls with their *CLayer* equivalents. For example, porting *wget* required changes in just 10 lines. Table 3 lists the source line count for the core *CLayer* implementation (C++) and extra lines (C/C++) required to port existing applications.

We used open-source *Google Protocol Buffers (protobuf)* [2] to encode/decode *CLayer* headers, instead of designing a customized header format. Although the dynamic nature of *protobuf* slightly increases header size and encode/decode complexity over a fixed width format, it greatly increased the pace and ease of prototyping. Note that the performance results obtained using this prototype implementation should be taken with a grain of salt as a lower bound of the per-

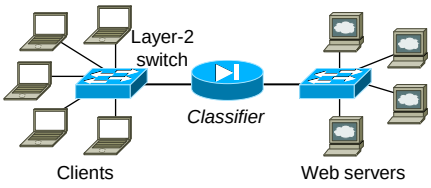

Figure 7: Topology used for firewall and load balancer performance evaluations.

formance of a deployment-ready implementation with optimizations.

6. EVALUATION

In this section we quantitatively evaluate how *CLayer* improves the scalability of classification dependent services using two examples: filtering using firewall and load balancing. The firewall example further illustrates how *CLayer* spurs classification offload in traditionally centralized applications and consequently improves performance.

6.1 Filtering using Firewall

The throughput of a regular firewall decreases with increasing rule set size [29]. Our *CLayer*-enabled firewall scalably maintains constant throughput that is two to three times that of a regular firewall at large rule sizes. We used the Click [19] `IPFilter` module as the base of our regular and *CLayer*-enabled firewalls. Firewall throughput was the sum of throughputs of simultaneous large file transfers between `wget` clients and `lighttpd` servers. Figure 7 shows our experimental topology created on the DETERlab [25] testbed.

We used two different rule sets in the firewalls under evaluation:

(i) `Snort`: Port number matches were sampled from the over 600 unique rule headers (i.e., involving just packet 5-tuples) in the Snort IDS rule set [8, 30]. Due to lack of IP diversity in the Snort rule set, source and destination IP matches were randomly drawn from a pool of 250 prefixes.

(ii) `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 the last. This enabled us to measure worst case performance, independent of the traffic mix.

Figure 8 and Figure 9 show the throughput drops of the regular firewall as rule set size increases from 100 to 4000 (error bars represent minimum and maximum values). For the `Snort` set, throughput drops more than 80% – ≈14.6MB/s → ≈2.5MB/s. For the `RandomIP` set, it drops around 60% – ≈13.6MB/s to ≈5.3MB/s. *CLayer*enabled firewalls maintain constant throughputs of ≈10MB/s and ≈12.5MB/s for the `Snort` set and the `RandomIP` set, respectively.

A *CLayer*-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

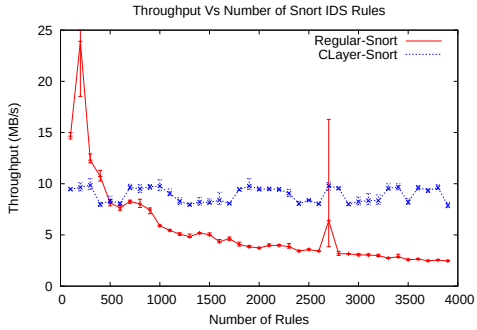

Figure 8: Firewall throughput versus number of rules using the `Snort` rule set.

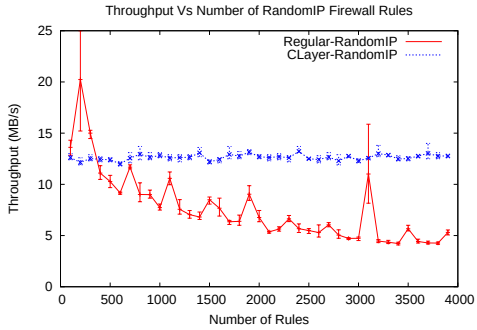

Figure 9: Firewall throughput versus number of rules using the `RandomIP` rule set.

scalability. In our experiments, the attractiveness threshold is around 500 and 375 for `Snort` and `RandomIP` rule sets respectively. Many firewall deployments already have rule sets that are larger than our thresholds. A 2004 study [26] found that firewalls have upto 2671 rules. The biggest classifier in [15] had 1733 rules, while the biggest edge router ACL set in [22] had 4740 rules. We expect rule set size to continue to grow as size and complexity of networks increase. Thus, attractiveness of a *CLayer*-enabled firewall is most likely to increase over time.

6.2 Load Balancing

Performance improvement in a *CLayer*-enabled load balancer from a regular load balancer is not as prominent as in case of firewalls. This stems from the fact that packet classification in firewalls is inherently more complex, and thus *CLayer* gets more opportunity for improvement. For preliminary evaluation, we developed a simple round-robin `LB` module in Click as the base of our regular and *CLayer*-enabled load balancers. Load balancer performance was measured in terms of average connection acceptance ratio and average connection handling time by using `httperf` benchmarking tool in clients and `lighttpd` servers. We use the same topology as in Figure 7 for these experiments and vary connection initiation rates from 100 connections/second to 1000 connections/second.

As evident from Figure 10 and Figure 11, under increasing load the *CLayer*-enabled layer-4 load balancer prototype attains higher acceptance ratio and lower handling time per connection than its regular counterpart. At the highest load

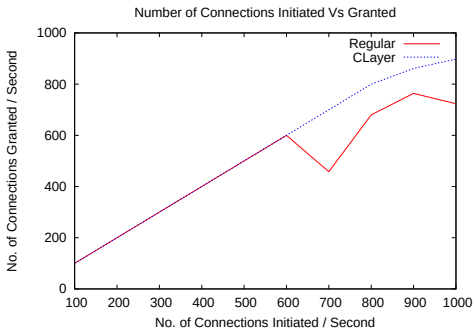

Figure 10: Load balancer average connection acceptance ratio.

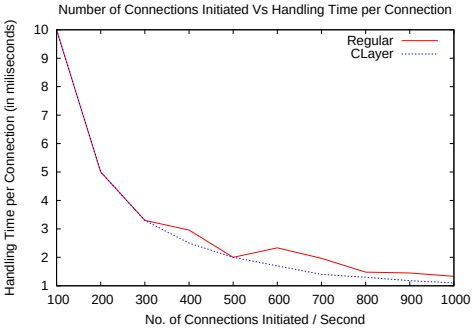

Figure 11: Load balancer average connection handling time.

in these experiments (i.e., 1000 connections/second) the regular load balancer can handle around 20% less connection while taking around 20% more time per connection. Based on the divergent nature of these graphs, we expect to observe even more difference in performance as we keep increasing the load in our ongoing experiments.

7. LIMITATIONS

We present the limitations of *CLayer* in the following:

1. **Space Overhead**
   CLayer incurs per-packet space overhead which limits the number of *labels* a packet can carry and consequently the number of *classifier* a *helper* can support. This can give rise to complex situations, specially in an inter-domain scenario where *classifiers* in one domain can *selfishly* overwrite *labels* provided by *classifiers* in another domain when a packet reaches its maximum *label*-carrying capacity. Providing proper incentives to prevent such selfish behavior is an open challenge.

2. **Performance Overhead**
   Even with *FCLs*, *CLayer* introduce small performance overhead in reading *labels* because there is no fixed location for *labels* from different *classifiers*. We believe settling for hardware-optimized *label* lengths (i.e., multiple of 4 bytes) can be a solution to this problem. Making *labels* fixed-length might help even more. However, this is a minor concern since performance benefits due to complexity reduction enabled by *CLayer* can easily mask and offset such overheads.

3. **Applicability**
   In its current state, *CLayer* offers only limited benefits to: (i) applications where classification result accuracy is critical but very expensive to check (e.g., NIDS), (ii) applications which perform complex operations for some other purpose (e.g., URL based filtering) anyway and may be overloaded for classification, and (iii) applications where no *helpers* exist or are all disincentivized to participate.However, the overall reduction in configuration complexity enabled by *CLayer* still benefits the network.

8. RELATED WORK

Many prior work advocated distributing packet classification load across network entities. Unlike our generic multi-layer, multi-application approach, these mechanisms offer point, and often ad-hoc, solutions focusing on a particular type of classification and application.

In MPLS [6], Label Switch Routers in the network core offload expensive route lookup operations to Label Edge Routers. Ipsilon [3] flow-switching is more general than its successor MPLS and supports packet classification at the network and transport layers. *CLayer* supports classification across layers 2 to 7 and uses an in-band signaling protocol not restricted to adjacent nodes.

In Diffserv [1], end hosts or first hop routers classify packets and record the desired QoS in the IP header's DS field. In CSFQ [23], edge routers label packets of a flow based on its flow rate. The 20-bit *flow-id* IPv6 header field [4] provides a mechanism for end-hosts to uniquely identify a flow with any desired semantics. Core routers can provide differential QoS to packets based on their DS fields, labels, or flow-ids without performing expensive reclassification. However, unlike *CLayer*, it supports only one application at a time and does not provide any signaling mechanism to inform/configure the entities that use the flow-id field.

Although originally designed for offloading web-server states to end hosts, HTTP cookies are widely overloaded as a means to identify multiple TCP flows in an HTTP session. Unlike HTTP cookies and the OSI session layer, *CLayer* is not restricted to the application layer – it works across layers 2 to 7. In addition, *CLayer* makes the session id available to a load balancer in an easily readable packet header location, as opposed to performing deep packet inspection or application header parsing to read an HTTP cookie.

Some prior work (e.g., distributed firewalls [11], network exception handlers [17]) adopted an extreme approach of moving the entire application requiring packet classification to end hosts. Our work targets the more conventional and widely deployed scenario where an in-network entity (e.g., a router or a middlebox) is involved (often in a critical role) in implementing the functionality that requires packet classification. *CLayer* can be used to communicate the results of network exception handlers to on-path entities.

In the OpenFlow [7] architecture, packet classification is offloaded to a logically centralized controller. Based on the initial packets of a flow, the controller classifies packets and installs flow table entries at switches on the flow's network path. *CLayer*'s distributed approach avoids a classification choke point and a centralized point of failure. *CLayer* trades off per-packet overhead in middleboxes for the overhead of state establishment in end hosts at flow startup.

COPS [18] proposes *iBoxes* that classify a packet using

deep packet inspection and then summarize the results in an *annotation layer* within the packet. *CLayer* simultaneously supports a variety of classification applications, in addition to security. *CLayer* headers are similar to X-trace [14] annotations in that their semantics and can span multiple protocol layers.

Unlike active networking [24], *CLayer* carries non-executable opaque bags of bits whose semantics depend on the classification application to which they are directed. This more restrictive nature of the *CLayer* avoids the security risks of executing untrusted code, while still enabling end hosts to influence the fate of their packets within the network.

SIFF [27] and Visa protocols [13] use *CLayer*-like mechanisms to mitigate DDoS flooding attacks and to enable secure inter-organizational communications respectively. SIFF requires capability establishment in all the routers on a path for privileged traffic using a handshake protocol. In *CLayer*, only the interested network elements need to add labels. Visa protocols use Access Control Servers in each domain to establish "visa" for each flow that are stamped on each packet using strong cryptographic methods. *CLayer* does not require any external server, and it aims for performance and can provide strong security with very high probability.

CLayer signaling borrows ideas from protocols used in different applications, including ECN [9], MIDCOM [5], RSVP [10], TVA [28], and HTTP Cookies. Stateful Distributed Interposition (SDI) [21] and Causeway [12] 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 end hosts to maintain session information and embed *CLayer* headers. This simplifies *CLayer* implementation and deployability.

9. CONCLUSIONS

Packet classification plays a fundamental role in enabling a diverse range of protocols and network services including switching, forwarding, filtering, and load balancing. In this report, we have demonstrated how treating packet classification as a fundamental network primitive by using *CLayer* can reduce its implementation and configuration complexity. Moreover, *CLayer* improves flexibility, performance, and scalability of packet classification without sacrificing security and privacy of concerned entities.

To demonstrate the feasibility of this approach, we have prototyped *CLayer* using Click software modular router, and we have enabled a variety of applications and network services with *CLayer* functionalities. Based on our preliminary experience of using *CLayer*, we believe that it has enough potential to support future classification applications without introducing additional point solutions.

10. REFERENCES

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

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

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

[4] IPv6 Flow Label Specification. RFC 3697.

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

[6] Multiprotocol Label Switching Architecture. RFC 3031.

[7] OpenFlow. `http://www.openflowswitch.org`.

[8] Snort. `http://www.snort.org`.

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

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

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

[12] A. Chanda et al. Causeway: operating system support for controlling and analyzing the execution of distributed programs. In *HOTOS*, 2005.

[13] D. Estrin et al. Visa Protocols for Controlling Inter-Organizational Datagram Flow: Extended Description. Technical Report WRL 88/5, Western Research Laboratory, Dec 1988.

[14] R. Fonseca et al. X-Trace: A Pervasive Network Tracing Framework. In *USENIX NSDI*, 2007.

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

[16] Y. He et al. On routing asymmetry in the Internet. In *GLOBECOM 2005*.

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

[18] R. H. Katz et al. COPS: Quality of Service vs. Any Service at All. In *IWQoS*, 2005.

[19] E. Kohler et al. The Click modular router. *ACM TOCS*, 18(3):263–297, Aug 2000.

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

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

[22] S. Singh et al. Packet Classification Using Multidimensional Cutting. In *SIGCOMM 2003*.

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

[24] D. Tennenhouse et al. A Survey of Active Network Research. *IEEE Comm. Magazine*, Jan 1997.

[25] B. White et al. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI 2002*.

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

[27] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *In IEEE Symposium on Security and Privacy*, pages 130–143, 2004.

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

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

[30] F. Yu et al. SSA: a power and memory efficient scheme to multi-match packet classification. In *ANCS*, 2005.