Self-Verifying CSFQ

Ion Stoica, Hui Zhang, Scott Shenker

Abstract—

Recently, a class of solutions including Core-Stateless Fair Queueing (CSFQ), Rainbow Fair Queueing, and Diffserv have been proposed to address the scalability concerns that have plagued stateful architectures such as Intserv and Fair Queueing. However, despite some desirable properties, these solutions still have serious scalability, robustness, and deployment problems. Their scalability, suffers from the fact that the core cannot transcend trust boundaries (such as at ISP-ISP interconnects), and so the high-speed routers on these boundaries must maintain per flow or per aggregate state. The lack of robustness is because a single malfunctioning edge or core router could severely impact the performance of the entire network. The deployability is hampered because the set of routers must be carefully configured with a well-defined set of edge routers surrounding the core.

In this paper, we propose an approach to address these limitations. The main idea is to use *statistical verification* to identify and contain the flows whose packets carry incorrect information. To demonstrate the applicability of this approach we develop an extension of CSFQ, called Self-Verifying CSFQ (SV-CSFQ). With SV-CSFQ, rate estimation is performed by sending hosts, and all routers statistically verify these rate estimates. Statistical verification allows routers to identify misbehaving flows and routers, and thereby protect other flows. This makes our approach robust and highly scalable as it eliminates the need for stateful routers at trust boundaries, and for the core-edge distinction. We present simulations and analysis of the performance of this approach, and discuss its general applicability to provide other scalable and robust network services.

Keywords—**Quality of Service, Stateless Core, Scalability, Fair Queueing.**

I. Introduction

To alleviate scalability problems that have plagued per flow solutions such as Fair Queueing [1] and Intserv [2], two new architectures have been recently proposed: Differentiated Services (Diffserv)[3], [4] and Stateless Core (SCORE) [5], [6], [7], [8]. A key common feature of these architectures is the distinction between edge and core routers in a trusted network domain (see Figure 1). The scalability is achieved by not requiring core routers to maintain any per flow state, and by keeping per flow or per aggregate state *only* at the edge routers.

In particular, with Diffserv, edge routers maintain per aggregate state [3], [9], [4], [10]. Core routers maintain state only for a very small number of traffic classes; they do not maintain any fine grained state about the traffic. With SCORE, edge routers maintain per flow state and use this state to label the packets [5], [6], [7], [8]. Core routers maintain no per flow state. Instead they process packets based on the state carried by the packets, and based on some internal aggregate state.

While Diffserv and SCORE have many advantages over traditional stateful solutions, they also have serious limitations. The main source of these limitations is the implicit assumption that the information carried by the packets inside the network domain is "correct". This assumption is needed because core routers process packets based on this information. As a result, Diffserv and SCORE *cannot* transcend trust boundaries. This introduces three significant limitations.

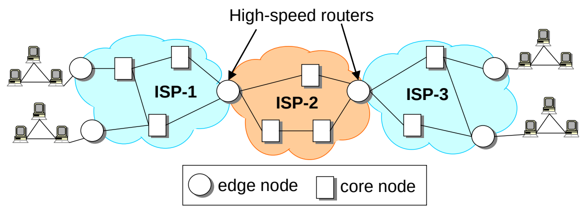


Fig. 1. Diffserv and SCORE network architectures. Edge routers maintain per aggregate or per flow state. Core routers maintain no per flow state. High-speed boundary routers between distinct trusted domains are edge routers.

First, these architectures are not very robust because core routers have to trust information carried by the arriving packets, such as flow rate estimates or the DS field content. This enables a single faulty router to disrupt the service in an entire core merely by inserting incorrect state.[1]

Second, precisely because malfunctioning routers can have such a severe impact it is unlikely that cores will transcend trust boundaries. In particular, for reasons of trust the high-speed border routers between ISPs will be edge, not core, routers (see Figure 1). Thus, many of the claimed scaling advantages for core-stateless architectures will not be realized because these high-speed border routers will be required to perform per flow operations and keep per flow state.

Third, the deployment and configuration of routers must be done with extreme care to ensure there is a well-defined set of edge routers surrounding the core. As noted above, a misconfigured router can have serious consequences at the level of the entire domain.

The goal of this paper is to address these limitations in the case of SCORE solutions. The main idea is to use *statistical verification* to identify and contain the flows whose packets carry incorrect information. To make this idea concrete, in this paper we focus on the case of CSFQ. In particular, we propose a design called *Self-Verifying* CSFQ (SV-CSFQ) that uses statistical verification to check the rate estimates. Routers no longer blindly trust the incoming rate estimates, instead they statistically verify and *contain* flows whose packets are incorrectly labeled. Thus, flows with incorrect estimates cannot seriously impact other traffic. Trust boundaries no longer require special treatment, and this allows us to move the burden of rate estimation all the way to the end-hosts. As a result, the design is robust, fully scalable, involves no configuration into separate core and peripheral regions.

It is important to note that what makes our approach possible in the case of SCORE solutions is that the state carried by flow

I. Stoica is with UC Berkeley. E-mail: istoica@cs.berkeley.edu
H. Zhang is with Carnegie Mellon University. E-mail: hzhang@cs.cmu.edu
S. Shenker is with Insitute for Internet Research at ICSI, Brekeley. E-mail:shenker@icci.berkeley.edu.

[1]For instance consider the premium service [11] provided by Diffserv. If an edge router misbehaves by letting extra premium traffic inside the network, this can compromise the premium service in the entire domain. At the limit, it can increase the loss rate of the premium traffic between any ingress-egress routers at intolerable levels.

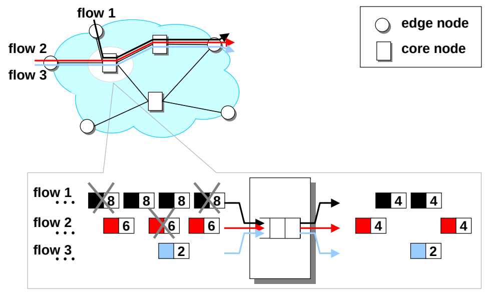


Fig. 2. Example illustrating the CSFQ algorithm at a core router. The crosses indicate dropped packets.

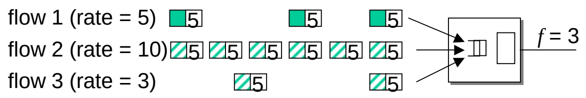


Fig. 3. Flow 1 is consistent, flow 2 is downward-inconsistent, and flow 3 is upward-inconsistent.

packets *describes* the flow behavior. For instance, in the case of CSFQ, packet labels carry the flow estimated rate. This makes the verification process easy, as routers can monitor the flow behavior, re-compute its state (e.g., re-estimate flow's rate in the case of CSFQ), and then compare the re-computed state against the state carried by flow's packets. In contrast, with Diffserv, it is much harder to verify the correctness of the DS field. While we believe that elements of our approach can be also used in the case of Diffserv, how to do it (e.g., whether we need extra information in the DS field) remains an open problem.

The remainder of the paper is organized as follows. Section II presents an overview of CSFQ, and discusses its limitations. Section III presents the assumptions and the properties of our solution, *Self-Verifying* CSFQ (SV-CSFQ). Section IV describes in detail SV-CSFQ, including its properties, complexity, and deployment aspects of SV-CSFQ. Section V presents detailed simulations in ns-2 with a variety of traffic types and topologies to illustrate the behavior of SV-CSFQ, and Section VI concludes the paper.

II. Background

This section gives an overview of CSFQ, and discusses the impact of incorrectly labeled packets on other traffic.

A. Core-Stateless Fair Queueing (CSFQ)

CSFQ [7] and its variants [5], [6], [12] aim to achieve fair bandwidth allocation in a highly scalable fashion. To achieve this goal CSFQ identifies an *island* of routers that represents a contiguous and trusted region of network, such as an ISP domain, and distinguishes between the edge and the core of the domain. Edge routers compute per-flow rate estimates and *label* the arriving packets by inserting these estimates into each packet header. Core routers use FIFO queueing and keep no per-flow state. They employ a probabilistic dropping algorithm that uses the information in the packet labels along with the router's own measurement of the aggregate traffic. Upon the arrival of a packet p, a core router enqueues the packet with probability

$$p_{enq} = \min\left(1, \alpha / p.label\right), \qquad (1)$$

and drops it with probability $p_{drop} = 1 - p_{enq}$, where *p.label* denotes the packet label, and α represents the max-min fair rate on the output link. If all packets of flow i are labeled with the flow's rate, r_i, flow i will receive on the average $\min(r_i, \alpha)$ bandwidth, which is exactly the fair bandwidth allocation of the flow. Given n flows that traverse a congested link of capacity, C, the fair rate α is defined by $\sum_{i=1}^{n} \min(r_i, \alpha) = C$ [7].

Figure 2 shows an example in which three flows with incoming rates of 8, 6, and 2, respectively, share a link of capacity 10. Note that in this case $\alpha = 4$. From Eq. (1), it follows that the forwarding probabilities of the three flows are 0.5, 0.66, and 1, respectively. Thus, flows 1 and 2 are allocated a rate of 4, and flow 1 is allocated its incoming rate of 2. To reflect the change of flow rates due to packet dropping, the router updates the rate estimates in the packet headers of flows 1 and 2 to 4.

B. Inconsistent and Consistent Flows

A flow whose packets carry inconsistent state is called *inconsistent*. A packet is said to carry inconsistent state if its label is different from the flow's rate. There are two types of inconsistent flows. If the label of a packet is larger than the actual flow rate, the flow is *upward-inconsistent*, and if the label is smaller than the flow rate, the flow is *downard-inconsistent* (see Figure 3). As we will show in the next section, of the two types of inconsistent flows, the downward-inconsistent ones are more dangerous as they can steal bandwidth from the consistent flows. In contrast, upward-inconsistent flows can only hurt themselves.

C. The Impact of Inconsistent Flows: An Example

This section discusses the impact of a downward-inconsistent flow on the consistent traffic, and the impact of a misbehaving router on the traffic in the entire domain.

Consider the example in Figure 2 with the difference that the 8 Mbps flow is downward-inconsistent, i.e., its packets carry an inconsistent rate of 1 Mbps instead of 8 Mbps. As a result, the inconsistent flow will get an unfair advantage by receiving 8 Mbps, eight times more than the other two flows! Intuitively, this is because a core router cannot differentiate – based only on packet labels – between the case of an 8 Mbps inconsistent flow, and the case of 8 consistent flows sending at 1 Mbps each. CSFQ implicitly assumes that the information carried by all packets is consistent, and, as a result, it will compute a fair rate of 1 Mbps instead of 4 Mbps, the inconsistent flow receiving 8 Mbps.

Worse yet, a misbehaving router will affect not only the flows that traverses it, but *all* flows that share paths with the inconsistent flows that it generates. In contrast, in a stateful network, in which each router implements Fair Queueing, a misbehaving router can hurt *only* the flows it forwards.

III. Assumptions and Properties

In this paper, we focus on misbehaviors that make the stateless solutions less robust than stateful solutions. In particular, we focus on packet labels being corrupted. We do not explicitly address misbehaviors for which stateful solutions are equally vulnerable, such as IP address spoofing or a user gaining unfair advantage by opening a large number of parallel connections.

However, we note that even in the presence of these misbehaviors our solution may perform better than stateful solutions such as Fair Queueing, since by containing the bad flows, it makes more bandwidth available to well-behaved flows.

We assume that a *flow* is defined by a subset of fields in the IP header. Without loss of generality, in this paper, we assume that a flow is defined by the source and destination IP addresses.

Our solution has the following goals and properties:

(1) The main goal of a router in real time is to protect the consistent flows, rather than identifying and containing misbehaving flows. This allows us to optimize the algorithms for identifying "large" inconsistent flows which have the most impact on the well-behaved traffic. Catching small inconsistent flows that have a negligible impact on the well-behaved flows is only of secondary importance. In particular, inconsistent flows whose arrivals rate are less than the fair rate have *no* effect on the other traffic.

(2) In a well engineered system, access routers end up containing most of the misbehaving flows. This assumes that access routers have enough resources to contain all inconsistent flows. We believe this a reasonable assumption given the fact that access routers have much lower capacity, and are traversed by much fewer flows[2] than core routers.

(3) If the number of misbehaving flows traversing a router exceeds the number of flows the router can contain[3], the router's primary goal is to protect the *down-stream* traffic. The router may take this action even if this means hurting well-behaved flows that traverse that router. The goal is to protect the rest of the network from a region "infected" with many misbehaving flows.

(4) Once a router identifies an inconsistent flow, it thereafter contains it. The containment procedure is orthogonal to the identification mechanism. For instance, the goal of containment can be to compensate for the extra service received by the inconsistent flow before being caught, or to severely punish the flow to give users incentive to well-behave. At the limit an inconsistent flow can be blocked.

(5) Each router operates independently, and does not make any assumption about the algorithms implemented by other routers. This makes our architecture both robust and incrementally deployable.

Given property (2) and that the spectrum of usage in the Internet is heavy-tailed [14], we expect a core router to be traversed by relatively few "large" inconsistent flows. Since our main focus is to detect and contain these flows, core routers don't need to maintain large amounts of state.

Finally, it might be useful to have an additional mechanism that retroactively identifies low-bandwidth/short-live misbehaving flows. This allows us to eventually identify a malicious user, and punish it at a later time (e.g., by administrative means). One way to implement such a mechanism is to inspect traffic logs off-line; however, the design of such mechanisms is orthogonal to this paper.

[2]According to recent data, access routers handle $\leq 5,000$ flows [13].
[3]This can happen in the case of a denial of service attack, or when access routers do not deploy our solution, or upstream routers are faulty.

IV. Self-Verifying CSFQ (SV-CSFQ)

This section presents a solution that addresses the limitations of CSFQ, called *self-verifying* CSFQ (SV-CSFQ). At the heart of our solution is the router ability to *identify* and *contain* inconsistent flows. Since only downward-inconsistent flows can deny service to other flows, our focus is to identify those flows. Unless otherwise specified, in the remainder of this paper, we will refer to downward-inconsistent flows simply as *inconsistent* flows.

The ability of identifying inconsistent flows removes the need of trusting the state in the packet headers. Thus, the edge is no longer confined to the trusted boundaries. At the limit, the edge can be pushed all the way to the end-hosts. In this case, our solution no longer differentiates between edge and core routers, so it eliminates the need for router configuration. In addition, the ability to transcend trusted boundaries makes our solution highly scalable, while the flow identification and containment makes it robust.

To identify inconsistent flows, SV-CSFQ routers implement *flow verification*. Flow verification consists of three phases: (1) randomly select a flow, (2) re-estimate the flow's rate, and (3) check whether the re-estimated rate is within a predefined threshold from flow's packet labels. If yes, the flow is declared consistent; otherwise the flow is declared inconsistent. If a flow is declared inconsistent, the router starts to contain that flow. Flow containment involves relabeling the flow packets with a label that exceeds the actual flow rate. This guarantees that a contained flow will receive no more than its fair rate.

Containing a flow has two immediate implications. First, the flow will no longer affect consistent traffic at downstream routers. Second, if the flow was contained by a downstream router, that router will eventually release the flow once it notices that the flow has became consistent. As a result, in steady state, only access (edge) routers will maintain state for *inconsistent* flows. To see why, consider an inconsistent flow that traverses two routers A and B, in this order. Assume that B identifies the flow first. After some time, A also identifies the flow and contains it. By containing the inconsistent flow, A transforms the inconsistent flow into a consistent one. In turn, as soon as it notices that the flow is no longer inconsistent, B will release the state of the flow. Thus, in the end, A will be the only one to contain and maintain state for the inconsistent flow.

The pseudo-code of the flow verification algorithm is shown in Figure 4. Each router maintains two separate tables per output interface: one for the verified flows, called (*VerifyTable*), and one for the contained flows, called (*ContainTable*). Upon a packet arrival, the router checks whether the packet belongs to a flow that is already being verified or contained. If not, and if there is room in *VerifyTable*, the router decides to start verifying the flow with probability p. In our implementation, we set p to 0.5. In practice, every router can choose p independently. This randomness makes it very difficult, if not impossible, for an attacker to evade the verification algorithm. If the router decides to verify a flow, it creates an entry for that flow in *VerifyTable*. If the flow is already being verified, the router checks whether the flow is consistent. If yes, the flow is removed from *VerifyTable*. If the flow is inconsistent, the flow is transferred

```
on receiving packet $p$
  $f = get\_flow\_filter(p)$;
  if $(f \notin VerifyTable$ and $f \notin ContainTable)$
    // if still room in VerifyTable, select f randomly
    if $(!full(VerifyTable)$ and $(\mathbf{random}(0,1) < p))$
      insert$(VerifyTable, f)$;
      initialize state of flow $f$;
  else // f is either verified or contained
    $f.rate = estimate\_rate(f.rate, p)$;
    if $(f \in VerifyTable)$
      if (is_consistent$(f)$)
        remove$(VerifyTable, f)$;
      if $(is\_inconsistent(f))$
        remove$(VerifyTable, f)$;
        insert$(ContainTable, f)$;
        initialize state of flow $f$;
    else // $f \in ContainTable$
      if $(is\_consistent(f))$
        remove$(ContainTable, f)$;
      else
        contain_flow$(f)$;
        // make the flow consistent
        $p.label =$ update_rate$(f.rate, p)$;
        return;

    // CSFQ code executed by core routers (see [7])
    $p_{drop} = \mathbf{max}(0, 1 - \alpha/p.label)$;
    ...
```

Fig. 4. Pseudo-code of SV-CSFQ.

from *VerifyTable* to *ContainTable*.[4] Once a flow is contained, the router limits its service rate, and updates the labels in its packets. If a contained flow becomes consistent again, the flow is removed from *ContainTable* after some period of time.

The next two sections discuss the two building blocks of SV-CSFQ: flow verification and flow containment. We emphasize that here that we discuss *one* possible realization of the pseudo-code in Figure 4. We expect that the verification and containment algorithms to significantly improve over time. However, as suggested by the simulation results presented in Section V, even in their current form, these algorithms are effective in protecting the consistent traffic.

A. Flow Verification

In the case of SV-CSFQ, the goal of flow verification is to identify flows whose packet labels are smaller than the actual flow rate. Here, we consider the following simple identification test. Once a flow f is inserted in *VerifyTable*, we monitor the flow for a predefined interval of time T_{ver}, called verification interval. During this interval, the router uses an algorithm identical to the one employed by CSFQ to perform rate estimation. The only difference is that we initialize the estimated rate to the label of the first packet of the flow, i.e., the packet that has triggered the verification.

At the end of the verification interval, the router uses the estimate of the flow rate and the incoming packet labels to decide whether the flow is consistent or not. Upon the arrival of the first packet p of flow f after T_{ver} expires, we compute the relative discrepancy of flow f as:

$$dis_{real} = (f.rate - p.label)/p.label, \qquad (2)$$

where $f.rate$ represents the estimated rate of f at the current router, and $p.label$ denotes the label of packet p. Then, we classify a flow as inconsistent if its relative discrepancy exceeds a predefined threshold, H_u. Otherwise, we classify the flow as consistent. If the flow is classified as consistent, it is deleted from *VerifyTable*. If the flow is classified as inconsistent, it is inserted into *ContainTable*.

In the technical report [15], we justify this identification test by using a simplified buffer-less router model, and discuss in detail the trade-offs in choosing the two parameters of the verification algorithm: T_{ver} and H_u. In short, while a large T_{ver} increases the accuracy of the rate estimation and therefore of the test accuracy, it delays the identification of inconsistent flows. This will increase the damages inflicted by the inconsistent flows before being caught. Similarly, while a large H_u reduces the probability of misidentifying a consistent flow, it also increases the probability that inconsistent flows will escape the test. To summarize our findings in [15], we choose $H_u = 0.2$, and T_{ver} as the time required to receive ten packets of the flow.

Finally, note that while our verification test may not catch very short flows, we believe this is an acceptable behavior as very short flows gain little from being inconsistent (see Section V-C), and their impact on other traffic is minimal. In addition, note that although small flows represents the majority of flows in the Internet, the traffic volume is still dominated by "large" flows [14].

B. Flow Containment

Once a router identifies an inconsistent flow, it starts to contain it. Recall that when a flow is contained, the router restricts its allocated rate. Any flow containment algorithm needs to answer two questions: (1) by how much is the rate of a contained flow restricted, and (2) what is the minimum interval of time for which a flow is contained.

Before answering these questions, we make one point. The algorithm described in this section is aimed to compensate for the excess bandwidth received by an inconsistent flow, rather than punishing that flow. This policy is appropriate for core routers inside a domain whose edge routers implement SV-CSFQ. This would avoid overreacting to edge routers misbehaviors. However, one might argue that this is not enough for deterring users from misbehaving. We can easily fix this by having the router drop more aggressively the packets of an inconsistent flow (i.e., use a much larger penalty factor in Eq. (5)). At the limit the router can simply bar the inconsistent flows.

B.1 Restricting Rate Allocation of Inconsistent Flows

To compensate for the extra service that an inconsistent flow may have received before being caught, we have to restrict the

[4]Note that a verified flow may be neither consistent or inconsistent. This is because, as we will discuss in the next section, we need to monitor the flow for a minimum interval of time before we can decide whether the flow is consistent or not. If we test a flow too early, the inaccuracies in the rate estimation can lead to misclassifying a consistent flow as being inconsistent.

rate allocated to a contained flow to less than the fair rate on the output link α. Similarly to CSFQ, we restrict the rate allocated to a contained flow f by dropping each of its incoming packets, p, with probability

$$p_{drop} = \max(0, 1 - (\alpha/(f.k_{pen} * p.label))), \quad (3)$$

where $f.k_{pen} \geq 1$ is called *penalty* factor. Note that if $f.k_{pen} = 1$, the dropping probability computed by Eq. (3) is exactly the dropping probability used by CSFQ [7].

Let $f.rate$ be the estimated arrival rate of flow f. The net effect of dropping each incoming packet, p, with probability p_{drop} is to limit f's rate allocation to

$$f.rate * (1 - p_{drop}) = \min\left(f.rate, f.k_{inc} * \alpha/f.k_{pen}\right), \quad (4)$$

where $f.k_{inc} = f.rate/p.label$ denotes the flow inconsistency factor measured upon p's arrival. In our current implementation, we choose

$$f.k_{pen} = (f.k_{inc})^2. \quad (5)$$

This restricts the rate allocated to flow f to $\min(f.rate, \alpha/f.k_{inc})$. Thus, while with CSFQ a misbehaving flow gets more bandwidth than a consistent flow, with SV-CSFQ, once the flow is contained, it gets less.

Our choice of $f.k_{pen}$ has two other implications. First, if a flow is erroneously classified as inconsistent, the flow is just slightly punished, as most likely such a flow has $f.k_{inc} \simeq 1$. Thus, occasional misclassifications of a consistent flow as inconsistent have little impact on its end-to-end performance. Second a flow that misbehaves in a big way, that is, a flow f for which $f.k_{inc} \gg 1$, is severely punished. This is intended to discourage a flow of intentionally misbehaving.

Finally, to account for the fact that flow parameters change over time, in practice we choose $f.k_{pen}$ as being

$$f.k_{pen}(t) = f.k_{inc}(t) * \max\left(f.k_{inc}(t), f.k_{inc}(f.stime)\right), \quad (6)$$

where $f.stime$ represents the time when f starts to be contained. The reason for which we take the maximum between $f.k_{inc}(t)$ and $f.k_{inc}(f.stime)$ is to avoid the cases in which f may escape being punished by suddenly starting to send consistent traffic once it detects that it has been contained.

B.2 Minimum Containment Interval Length

As discussed in the previous section, as long as an inconsistent flow evades the verification test it gains unfair service. If the time interval during which a flow is contained is too small, a malicious[5] flow can take advantage of it. In particular, a malicious flow can start sending consistent traffic once it detects that it was caught[6] to minimize the time while it is being contained. If the service gained before being caught is larger than the service lost during the containment period, the malicious flow will gain service overall.

Our main goal in setting the minimum time interval for which a flow is contained, also called the minimum containment interval, is precisely to avoid this possibility. Assume an inconsistent flow f characterized by an inconsistency factor $f.k_{inc}$ that traverses a congested link with fair rate $\alpha < f.rate$. For simplicity, assume that α and $f.k_{inc}$ do not change over the time, i.e., each packet of flow f will carry the label $f.rate/f.k_{inc}$. Thus, as long as the flow is not caught, it will receive $f.k_{inc} * \alpha$ bandwidth, instead of α. Assuming that f is not caught for t_1 time, f will receive

$$t_1 * (f.k_{inc} - 1) * \alpha \quad (7)$$

excess service as compared to a consistent flow. In contrast, as described in the previous section, after f is caught, its bandwidth is restricted to $\alpha/f.k_{inc}$. Assume the flow is contained for time $f.contain\_int$. During this time interval the flow receives

$$f.contain\_int * (1 - 1/f.k_{inc}) * \alpha. \quad (8)$$

less service than an equivalent consistent flow.

We then choose $f.contain\_int$ such that the service lost by flow f during the time it was contained is at least equal to the excess service it has gained before being caught, i.e., $f.contain\_int \geq k_{inc} * t_1$. This will ensure that flow f will not get on the average more bandwidth than a consistent flow sending at the same rate.

There is still one problem in choosing a value for $f.contain\_int$: we do not know the value of t_1. However, since at the minimum t_1 is the time it takes to verify a flow for consistency, in our implementation we simply choose:

$$f.contain\_int = f.k_{inc} * T_{ver}. \quad (9)$$

While there is a probability that a malicious flow may receive a little extra service, we believe this choice strikes a good balance between the two conflicting goals of (1) punishing malicious flows and (2) minimizing the time for which an occasionally misidentified flow is contained.

B.3 Aggregate Containment

While in most cases we expect that the number of inconsistent flows to be smaller than the *ContainTable* size, there can be cases in which this is not true. Two relevant examples are: (a) hosts attacking edge routers by sending many inconsistent flows, and (b) an internal router failing and mislabeling (all) flows that pass through it. There are at least two possible actions we can take in this case. First, and the easiest is to alert the network administrator. Second, is to switch from the model of containing only individual flows to contain the *entire* traffic. The motivation behind the second approach is discussed in Section III: if there are too many offenders in a particular network region, the main goal is to protect the rest of the network from that region. Next, we discuss this approach in more detail.

Once *ContainTable* is full, we use the smallest inconsistency factor k'_{inc} among all flows in *ContainTable* to "contain" *all* the other flows. In particular, we multiply the label of each packet of an un-contained flow (i.e., of a flow not in *ContainTable*) with k'_{inc} before enqueueing it. Here, we assume that a router always contains the flows with the highest inconsistency factors detected so far[7]. As a result, the inconsistency factor of an un-

[5]A *malicious* flow is a flow sent by an adversarial source that tries to defeat our verification and/or containment mechanisms.

[6]Here we assume that the flow *knows* when it was caught. In practice this is difficult, which makes it even harder for a malicious flow to gain unfair service.

[7]The reason for choosing this policy is driven by the assumption that the flow inconsistency factor is roughly correlated to the flow bandwidth. Thus, because *ContainTable* will end up containing the highest bandwidth flows, the rest of the flows will have an inconsistency factor no larger than the smallest inconsistency factor of any flow in *ContainTable*. While the correlation assumption is not true in general, we use it here as a first order approximation.

contained flow is unlikely to exceed k'_{inc}.

Thus, by multiplying the labels of all packets belonging to un-contained flows with k'_{inc}, we reduce the damage inflicted by the un-contained inconsistent flows at the downstream routers. A potential problem though is that multiplying the labels of a consistent flow with k'_{inc} will make that flow *upward* inconsistent. As a result, such a flow might be unfairly penalized at a downstream router (see Section II-C). In practice this problem is alleviated by the fact that the probability that a flow will traverse another router with the same fair share is small.

C. Properties of Flow Verification

This section presents two key properties of our solution, which are proven in [15]. The first result estimates the time to catch an inconsistent flow (see Theorem 1). Theorem 2 gives an estimate of the total service allocated to inconsistent flows before all of them are caught and contained. Since an inconsistent flow cannot steal more service from the consistent traffic than it receives, this results bounds the damage inflicted by the inconsistent traffic on the consistent traffic.

Theorem 1: Assume the fraction of the inconsistent traffic is f_{inc}, and the size of *VerifyTable* is M. Then, the expected time until the *first* inconsistent flow is caught is:

$$T_{ver}/(M * f_{inc}) + T_{ver}. \tag{10}$$

The first term is the expected time to select an inconsistent flow to be verified. This time is inverse proportional to the *VerifyTable* size (a larger table allows for more flows to be monitored in parallel), and to the fraction of the inconsistent traffic f_{inc} (a higher f_{inc} increases the probability to select an inconsistent flow to be monitored). The second term is the time it takes to verify the flow.

A more general result of how long it takes to catch *all* inconsistent flows is, unfortunately, more difficult to give, as it depends on the bandwidth of each inconsistent flow. However, we can give such a result for a particular case.

Corollary 1: Assume K inconsistent flows with equal bandwidth. Let f_{inc} be the fraction of bandwidth of all these flows. If each inconsistent flow is contained as soon as it is caught, the expected time to catch all inconsistent flows is

$$(T_{ver} * K * \log(K))/(M * f_{inc}) + T_{ver}. \tag{11}$$

The next result bounds the damage inflicted by inconsistent flows before being caught.

Theorem 2: Assume K inconsistent flows that arrive at a router at the same time t, and assume that there are no other inconsistent flows arriving before or after time t. Let f_{inc} be the fraction of the aggregate inconsistent traffic at time t. Then, if the size of *ContainTable* is at least K, the expected service allocated to inconsistent flows before *all* of them are caught is

$$C * T_{ver} * (K/M + f_{inc}), \tag{12}$$

where C is the link capacity.

The term $C * T_{ver} * f_{inc}$ accounts for the worst case scenario in which during the interval $[t, t + T_{ver})$ only flows selected before time t are being verified. Excepting this correction factor, note that the amount of service allocated to inconsistent flows does *not* depend on the throughput of these flows. This is because, while a high bandwidth inconsistent traffic may inflict more damage per time unit, it also allows us to catch inconsistent flows faster (see Theorem 1). In addition, as expected, the bound is inversely proportional to the number of flows that can be verified in parallel M, and directly proportional to the number of inconsistent flows K.

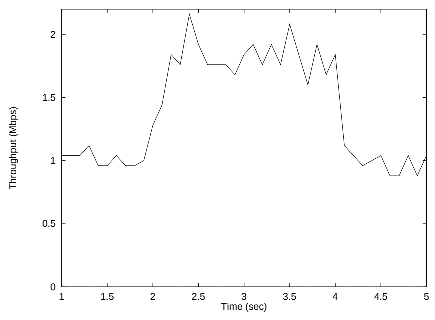
D. SV-CSFQ Complexity

In comparison to a *core* CSFQ router, a SV-CSFQ router has to perform three additional functions: (1) packet classification, (2) flow verification, and (3) flow containment. We argue that this extra complexity is manageable. This complexity depends strongly on the sizes of *VerifyTable* and *ContainTable*.

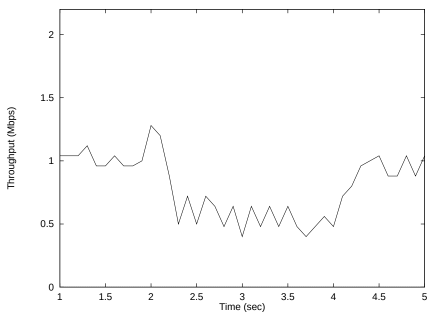
The *ContainTable* size, L, determines the maximum number of inconsistent flows that can be simultaneously contained. In turn, the size of the *VerifyTable*, M, determines how long it takes to catch an inconsistent flow (see Theorem 2). As discussed below, we expect that in most common cases the *ContainTable* size, L, to be larger than the total number of inconsistent flows K. Then, by simply choosing $M = K$, we can guarantee that the expected service lost to the inconsistent flows is less than $C * T_{ver} * (1 + f_{inc})$ (see Theorem 2). Note that this is within a factor of two of the best we can do in the worst case scenario when $f_{inc} \rightarrow 1$. Indeed, it takes the router at least T_{ver} to catch all the inconsistent flows, interval during which the inconsistent flows are allocated approximately $T_{ver} * C$ service.

Packet classification depends strongly on the number of filters, i.e., the number of flows that are classified [16]. Since a SV-CSFQ router needs to maintain filters only for flows in *VerifyTable* and *ContainTable*, and since, as discussed in Section III, these tables are small, we can achieve high speed classification by using any of the algorithms proposed recently [16], [17], [18], [19]. A simple hash table with the hash keys computed over the IP source and destination addresses would be sufficient for most practical purposes. Finally, flow verification and containment are constant time operations, and have a similar processing overhead as estimating the flow rate in CSFQ.

In conclusion, the overhead and the state maintained by a SV-CSFQ router is given roughly by the maximum number of inconsistent flows that we aim to contain, L. This appears to be a *fundamental limitation* of any algorithm that protects well-behaved flows by identifying and containing the misbehaving flows. This is because, at the limit, the router needs to maintain state for *each* flow it contains. Unfortunately, accurately predicting L is very difficult, as deploying SV-CSFQ will likely affect user behaviors, and traffic characteristics. Still, it appears that setting L to be of the order of the number of flows that traverse a typical access router is a good rule of thumb. First, such a value allows access routers to contain all flows, if needed. Second, this value is small enough (e.g., a couple of thousands [13]) to not compromise the scalability of core routers. We believe that using a relatively small value L for core routers is acceptable given SV-CSFQ's primary goal of containing "large" inconsistent flows, and the fact that SV-CSFQ can still protect the network in cases of DoS attacks by using aggregate containment.



(a)



(b)

Fig. 5. The throughput of a flow that misbehaves during the interval [2 sec, 4 sec) by sending inconsistent traffic with $k_{inc} = 2$ in the case of (a) CSFQ, and (b) SV-CSFQ, respectively.

E. Deployment Aspects

There are at least two possibilities to deploy SV-CSFQ. The first one is to start with neighbor domains that implement CSFQ, and use SV-CSFQ to remove high-speed edge routers at their boundaries (see Figure 1). This solves the CSFQ scalability problem, since now only access routers which run at a low speed have to maintain per flow state.

The second possibility, which we have considered in this paper, is to start with a clean-sheet design in which only end-hosts perform packet labeling, while routers implement SV-CSFQ. The advantage of this approach is that, in addition to achieving scalability and robustness, it also eliminates any need for configuration. However, if SV-CSFQ is to be adopted, it is critical to have a path that allows incremental deployment. We believe that in order to be successful, any such deployment path has to meet two criteria: (1) provide an incentive for end-hosts to label flow's packets, and (2) provide an incentive for ISPs to deploy SV-CSFQ routers.

One way to achieve these goals is to provide an additional router mechanism that makes sure that labeled flows receive "better" treatment than un-labeled flows. On one hand, this will give users incentive to label their packets because they will get better performances. On the other hand, ISPs will have incentive to deploy SV-CSFQ routers because they can provide better resource management and protection against malicious users.

A solution to provide "better" service to the labeled traffic is to implement aggregate containment for the un-labeled traffic, similar to the solution described in Section IV-B.3. Such a solution would statistically monitor un-labeled flows and use the highest flow rate that it sees to label *all* packets of the regular traffic. Developing such a mechanism remains a topic of future work.

V. Simulation Results

In this section, we evaluate SV-CSFQ. All simulations were performed in ns-2 [20]. Unless otherwise specified, we use the parameters recommended in [7]. Each output link has a buffer of 64 KB, and a buffer threshold of 16 KB. The averaging constant used in estimating the flow rate is $K = 100$ ms, while the averaging constant used in estimating the fair rate is $K_\alpha = 200$ ms. As an optimization, in our implementation we use only one table that is shared by both *VerifyTable* and *ContainTable*. The size of the shared table is six, and the maximum number of entries dedicated to the contained flows is four. Thus, a router can contain up to four flows, and verify up to six simultaneous flows, when no flow is contained. Routers select flows to be verified with probability 0.5. Each flow is verified for at least 200 ms or 10 consecutive packets, and we use a threshold H_u of 0.2. Please refer to [15] for the rationales behind these choices.

To simplify the notation, in the remainder of this paper, we drop f from $f.k_{inc}$ when this is clear from context.

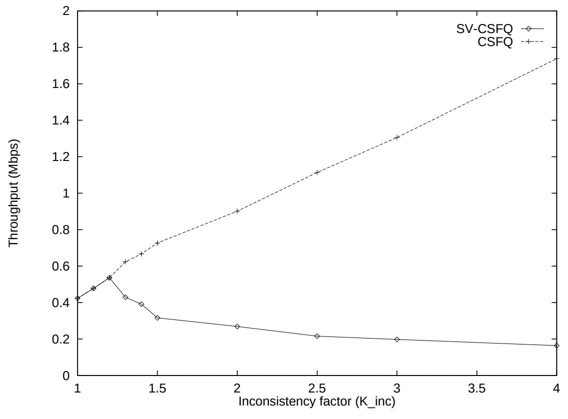
A. A Simple Example

In this section, we present a simple example to give an intuition of how SV-CSFQ behaves. Consider a 10 Mbps link traversed by 10 UDP flows. The arrival rate of each UDP is 2 Mbps, i.e., twice the fair rate of the link. Thus, the resulting fair rate is 1 Mbps. At time $t = 2$ sec, flow 1 starts to misbehave by sending inconsistent traffic with $k_{inc} = 2$, i.e., the labels carried by the flow's packets are half of the actual flow arrival rate. Finally, at time $t = 4$ sec, flow 1 starts to send consistent traffic again.

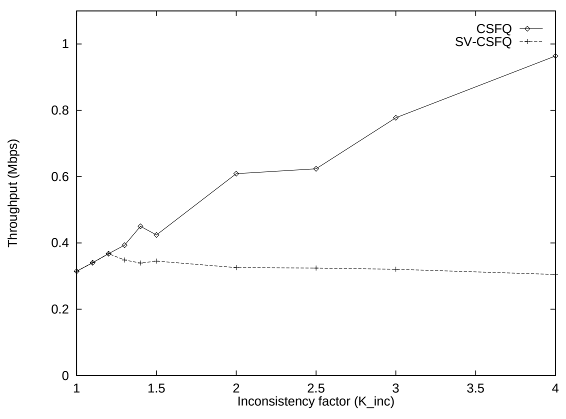
Figure 5 plots the throughput of flow 1 during the time interval $[1, 5)$ sec in the case of CSFQ, and SV-CSFQ, respectively. As expected, under CSFQ, when the flow starts to send inconsistent traffic with $k_{inc} = 2$ it gets approximately twice the fair rate, which in this case is 1 Mbps. In contrast, under SV-CSFQ, the inconsistent flow gets less than the fair rate (see Figure 5(b)). There are three distinct phases in the flow behavior. The first phase spans the time interval between $t = 2$ sec, when the flow start to misbehave, and $t = 2.24$ sec, when the flow is caught. As expected, during this time interval the flow gets considerable more bandwidth than its share. The second phase starts at $t = 2.24$ sec, and terminates at $t = 4.21$ sec, when the flow is released after it became consistent again. During this time, the flow is contained by restricting its rate to half (i.e., $1/k_{inc}$) of its fair rate (see Eqs. (4) and (5)). Finally, after the flow is released at $t = 4.21$ sec, it starts again to receive its fair rate.

B. The Impact of Inconsistency Factor, k_{inc}

The main parameter that characterize an inconsistent flow is its inconsistency factor, k_{inc}. In this section, we examine the impact of k_{inc} on the performance of SV-CSFQ. We consider a 10 Mbps congested link shared by 31 TCP consistent flows, and one inconsistent flow. Note that the fair rate on the congested link is about 0.3 Mbps.



(a)



(b)

Fig. 6. (a) The average throughput of an inconsistent UDP flow that share a congested link with 32 consistent TCP flows. (b) The average throughput of an inconsistent TCP flow with the same cross-traffic.

Figure 6 plots the average throughputs over a ten second interval when (a) the inconsistent flow is UDP, and when (b) the inconsistent flow is TCP. In both cases, when CSFQ is used, the inconsistent flow is allocated a rate that increases with its inconsistency factor. One interesting point to note is that the throughput of the TCP flow does not increase as fast as the UDP flow. This is because the inconsistent TCP will still end up losing consecutive packets, which will trigger retransmission timeouts.

In contrast, under SV-CSFQ, the throughput of the inconsistent flow remains close to its fair rate. There are two points worth noting. First, the throughput of the inconsistent UDP/TCP flow actually increases for small values of k_{inc}. This is because our current verification test does not classify a flow as inconsistent if the flow's measured inconsistency factor is ≤ 1.2. This suggests a more sophisticated verification test in which we start with a large H_u, and then gradually decrease it. Theoretically, such a test would allow us to identify a flow with an arbitrary small k_{inc} given enough time, and, at the same time, identify flows with large k_{inc}'s very fast. Such a verification test remains subject for future work. Second, note that as the inconsistency factor of the flow increases past 1.2, the actual rate allocated to the flow decreases. This is the result of limiting the rate of the inconsistent flow inversely proportional to its inconsistency factor (see Section IV-A).

C. Very Short Flows

Since flow verification requires each flow to be monitored for at least T_{ver} time, very short inconsistent flows may be never caught. However, as we have stated in Section IV-A, we believe that this is not a problem in practice, as very short flows will gain very little from being inconsistent, and will have minimal impact on other traffic. To support this point, we perform two experiments.

In the first experiment, we consider a short TCP flow f sharing a 10 Mbps link with other 32 long-lived TCP flows which are all well-behaved. Flow f transfers a fixed amount of data in 1 KB packets. For each transfer size, we repeat the experiment 20 times and we measure the transfer time of f. The first five rows in Table I show the mean transfer times and the standard deviations for three simulation scenarios: (a) f is consistent; (b) f is inconsistent and the containment mechanism is disabled (this is equivalent with running CSFQ); and (c) f is inconsistent. Scenario (b) is intended to illustrate how much f will benefit from being inconsistent. In cases (b) and (c) the inconsistency factor of f is four. We do not show the results for transfers involving less than eight packets as their transfer times are the same in all cases.[8] There are two points worth noting. First, even when f is not caught, it gets only minimal advantage, i.e., its transfer time improves by less than 3%. Second, as the size of the transfer increases, SV-CSFQ starts to heavily penalize flow f. In particular, when f has 128 packets, f's mean transfer time increases by 60%. This corroborated with the fact that the eventual payoff is minimal creates a natural incentive *against* sending short inconsistent flows.

Burst Length (packets)	Consistent Traffic	Inc. Traffic (not contained)	Inc. Traffic (contained)
8	1.571 (2.229)	1.571 (2.229)	1.586 (2.264)
16	1.822 (2.298)	1.796 (2.313)	1.840 (2.412)
32	1.585 (3.852)	1.534 (3.678)	1.562 (4.126)
64	4.269 (2.893)	4.177 (2.827)	5.173 (2.654)
128	6.424 (3.464)	6.368 (3.162)	10.268 (2.908)
Web Traffic	0.985 (3.067)	0.903 (1.484)	1.201 (4.127)

TABLE I
THE MEAN TRANSFER TIMES (IN SECONDS) AND THE STANDARD DEVIATIONS (IN PARENTHESIS) FOR SHORT TCP TRANSFERS.

The second experiment simulates Web traffic. Here, we consider 1000 short TCP transfers whose inter-arrival times are exponentially distributed with the mean of 0.1 ms, and the packet lengths are Pareto distributed with a mean of 20 and a shape parameter of 1.06 [7]. The cross traffic consists of 32 TCP flows. The last row in Table I presents the mean transfer times and the standard deviations. Again, even in the worst case scenario when none of the short TCPs is caught, the gain is less than 10%.

D. Multiple Congested Links

In this section, we illustrate the dynamics of flow containment algorithm in the case of multiple congested links. We consider

[8]This is because the transfers terminate before we get a chance to test the flows for consistency, before T_{ver} expires.

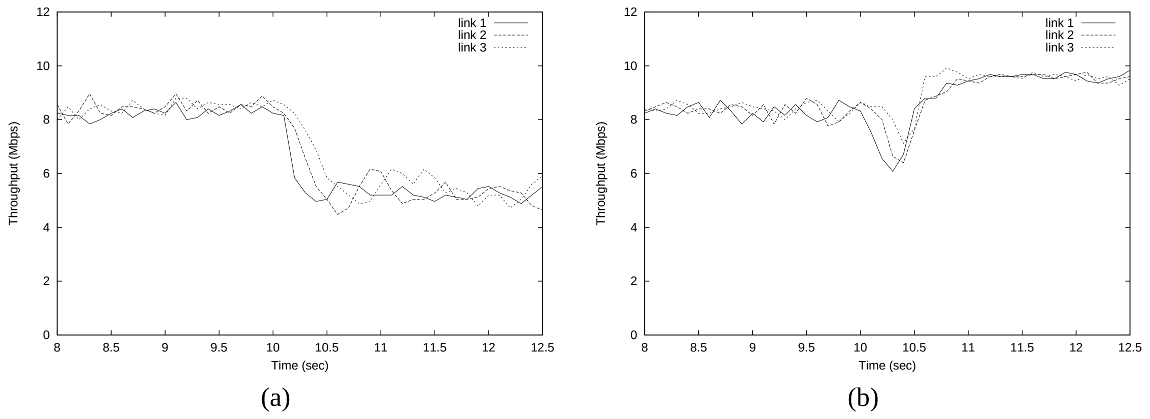


Fig. 8. The aggregate throughput of TCP cross-traffic at the congested links in the topology in Figure 7 when (a) baseline CSFQ is used, and when (b) SV-CSFQ is used. In both cases the UDP flows start to misbehave at time $t = 10$ sec.

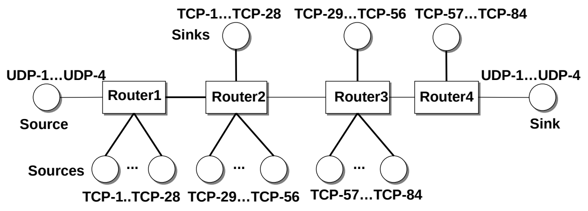


Fig. 7. "Multiple congested links" scenario topology.

the topology in Figure 7, where four 4 Mbps UDP flows traverse three 10 Mbps congested links. In addition, each congested link is traversed by a cross-traffic consisting of 28 consistent TCP flows. At time $t = 10$ sec, UDP flows start to misbehave by sending inconsistent traffic with $k_{inc} = 4$.

Figure 8 plots the aggregate throughput of the consistent traffic (i.e., of TCP flows) at each congested link, in the case of CSFQ, and SV-CSFQ, respectively. As expected, when UDP flows start to misbehave, the TCP throughput at each link drops significantly. Thus, each UDP will get roughly four times more than its fair rate.

In contrast, under SV-CSFQ, after an initial drop, the consistent traffic receives even more bandwidth (see Figure 8(b)). The initial drop is because it takes some time to catch all inconsistent UDP flows, and until then they will deny service to TCP flows. However, once UDP flows are identified and contained, the throughput of the TCP flows starts to climb up and stabilize at a value higher even than the value before $t = 10$ sec. Again, this is because once an inconsistent flow is detected, its rate is restricted to a value inversely proportional to its inconsistency factor k_{inc}.

Another interesting point to note is that immediately after $t = 10$ sec the TCP flows traversing the first link suffers the most, followed by the TCP flows traversing the second link, and then the third link. This is to be expected, as the probability of a flow not being caught decreases with the number of hops it traverses.

Finally, we note that by $t = 10.63$ sec, all UDP flows are contained at the first router, and by time $t = 10.98$ sec no other router excepting the first one maintains state associated to these flows. As an example, UDP-4 is caught by the third router at $t = 10.20$ sec, then by the second router at time $t = 10.45$ sec, and finally by the first router at $t = 10.67$ sec. The third router removes the state associated with UDP-4 at $t = 10.53$ sec, and the second router removes UDP-4 state at $t = 10.98$ sec. This example also illustrates two desirable properties of our algorithm. First, in steady state only one router maintains state for an inconsistent flow. Second, this router is actually the first router on the flow's path.

E. *Aggregate Traffic Containment*

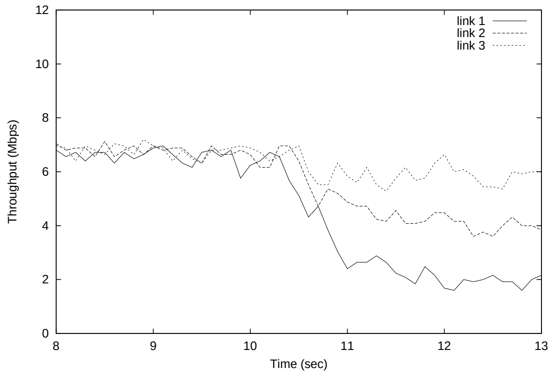
In this section, we illustrate the algorithm for aggregate traffic containment, as discussed in Section IV-B.3. We consider a similar simulation scenario as the one described in Section V-D. All links are traversed by 20 UDP flows, and the cross-traffic on each congested link consists of 60 consistent TCP flows. All UDP flows become inconsistent at $t = 10$ sec.

Figure 9 shows the aggregate throughputs of the consistent TCP flows on each congested link, in the case of (a) the baseline SV-CSFQ algorithm, and in the case of (b) the SV-CSFQ algorithm employing *aggregate containment* mechanism described in Section IV-B.3.

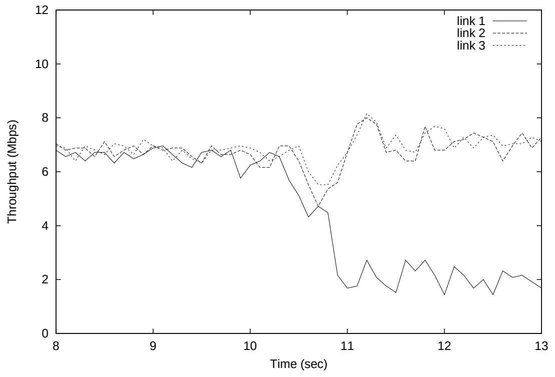
As expected, under the baseline SV-CSFQ, since the number of inconsistent flows is much larger than the *ContainTable* size, the first router can no longer contain all the flows. However, since every router along the path contains four flows each, the effect of the inconsistent traffic decreases as this advance into the network. As a result, the inconsistent flows inflict the most damage on TCP flows at the first link, and the least damage on the TCP flows that traverse the last link.

In contrast, when the aggregate containment is used, the consistent traffic on links 2 and 3 is protected. This is because, the first router updates the labels of all flows that are not contained to make sure that none of the UDP flows remains inconsistent. However, note that the consistent traffic on the first link is still affected. Again, this is because the first router can contain only four flows at a time.

Thus, while the aggregate traffic containment algorithm does not provide better protection at the first router, it protects the well-behaved traffic at subsequent routers.



(a)



(b)

Fig. 9. Example when number of inconsistent flows exceeds the size of *ContainTable*. The aggregate throughput of TCP cross-traffic at the congested links in the topology from Figure 7 for (a) the baseline SV-CSFQ scheme, and for (b) SV-CSFQ augmented with aggregate traffic containment algorithm.

VI. Summary

While SCORE based solutions such as CSFQ [7], TUF [6], and [8] are able to provide services equivalent to the ones implemented by stateful solutions in a more scalable fashion, they still suffer from robustness and scalability limitations. In particular, the scalability is hampered because the network core cannot transcend trust boundaries (such as the ISP-ISP boundaries), and therefore high-speed routers on these boundaries must be stateful edge routers. The lack of robustness is because the malfunctioning of a single edge or core router could severely impact the performance of the entire network, by inserting inconsistent state in the packet headers.

In this paper, we have proposed an approach to overcome these limitations. To achieve scalability we push the complexity all the way to the end-hosts. To address the trust and robustness issues, all routers statistically verify whether the incoming packets carry consistent state. This approach enables routers to discover and isolate misbehaving flows and routers. The key technique we use to implement this approach is flow verification that allows the identification of packets carrying inconsistent state. To illustrate this approach, we have described the identification algorithms in the case of Core-Stateless Fair Queueing (CSFQ), called Self-Verifying CSFQ and we have presented simulation results in ns-2 [20] to demonstrate its effectiveness.

Many more things remain to be done. First, the SV-CSFQ realization presented in this paper should be seen as a first iteration. We expect that current algorithms to be significantly improved in the future. One example is to use an adaptive verification interval and threshold to improve the identification accuracy. Another direction would be to explore how statistical verification approach works for other SCORE based solutions, such as TUF [6], and providing bandwidth and delay guarantees [8]. Finally, it would be interesting to see what elements if any from our approach can be used by Diffserv [3] architecture to increase its robustness.

References

[1] A. Demers, S. Keshav, and S. Shenker, “Analysis and simulation of a fair queueing algorithm,” in *Journal of Internetworking Research and Experience*, Oct. 1990, pp. 3–26.

[2] S. Shenker R. Braden, D. Clark, “Integrated services in the Internet architecture: An overview,” June 1994, Internet RFC 1633.

[3] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, “An architecture for differentiated services,” Dec. 1998, Internet RFC 2475.

[4] Juha Heinanen, F. Baker, W. Weiss, and J. Wroclawski, “Assured forwarding PHB group,” June 1999, Internet RFC 2597.

[5] Z. Cao, Z. Wang, and E. Zegura, “Rainbow fair queueing: Fair bandwidth sharing without per-flow state,” in *Proceedings of INFOCOM’99*, Tel-Aviv, Israel, Mar. 2000, pp. 922–931.

[6] A. Clerget and W. Dabbous, “Tag-based fair bandwidth sharing for responsive and unresponsive flows,” in *Proceedings of INFOCOM’01*, Anchorage, AK, Apr. 2001.

[7] I. Stoica, S. Shenker, and H. Zhang, “Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks,” in *Proceedings ACM SIGCOMM’98*, Vancouver, Sept. 1998, pp. 118–130.

[8] I. Stoica and H. Zhang, “Providing guaranteed services without per flow management,” in *Proceedings of ACM SIGCOMM’99*, Cambridge, MA, Sept. 1999, pp. 81–94.

[9] D. Clark and J. Wroclawski, “An approach to service allocation in the Internet,” July 1997, Internet Draft, http://diffserv.lcs.mit.edu/draft-clark-diff-svc-alloc-00.txt.

[10] K. Nichols, S. Blake, F. Baker, and D. L. Black, “Definition of the differentiated services field (DS field) in the IPv4 and IPv6 headers,” Dec. 1998, Internet RFC 2474.

[11] Van Jacobson and K. Poduri K. Nichols, “An expedited forwarding PHB,” June 1999, Internet RFC 2598.

[12] N. Venkitaraman, J. Mysore, R. Srikant, and R. Barnes, “Stateless prioritized fair queuing,” Aug. 2000, Internet Draft, draft-venkitaraman-diffserv-spfq-00.txt.

[13] C. Fraleigh, S. Moon, C. Diot, B. Lyles, and F. Tobagi, “Architecture for a passive monitoring system for backbone ip networks,” Oct. 2000, (submitted for publication).

[14] R. Mahajan, S. Floyd, and D. Wheterall, “Controlling high-bandwidth flows at the congested route,” in *Proceedings of IEEE ICNP’01*, Riverside, CA, Nov. 2001.

[15] I. Stoica, H. Zhang, and S. Shenker, “Self-verifying CSFQ,” http://www.cs.berkeley.edu/˜istoica/svcsfq-tr.pdf.

[16] Pankaj Gupta and Nick McKeown, “Packet classification on multiple fields,” in *Proceedings of ACM SIGCOMM’99*, Cambridge, MA, Sept. 1999, pp. 147–160.

[17] T.V. Lakshman and D. Stiliadis, “High speed policy-based packet forwarding using efficient multi-dimensional range matching,” in *Proceedings of ACM SIGCOMM’98*, Vancouver, Canada, Sept. 1998, pp. 203–214.

[18] V. Srinivasan, S. Suri, and G. Varghese, “Packet classification using tuple space search,” in *Proceedings of ACM SIGCOMM’99*, Cambridge, MA, Sept. 1999, pp. 135–146.

[19] V. Srinivasan, G. Varghese, S. Suri, and M. Waldvogel, “Fast scalable algorithms for level four switching,” in *Proceedings of ACM SIGCOMM’98*, Vancouver, Canada, Sept. 1998, pp. 191–202.

[20] “Ucb/lbnl/vint network simulator - ns (version 2),” http://www-mash.cs.berkeley.edu/ns/.