Loss and Delay Accountability for the Internet

Katerina Argyraki, EPFL, Switzerland
Petros Maniatis, Intel Research Berkeley
Olga Irzak, EPFL, Switzerland
Subramanian Ashish, EPFL, Switzerland
Scott Shenker, UC Berkeley

15th IEEE International Conference on Network Protocols, Beijing, China. October, 2007.

Abstract— **The Internet provides no information on the fate of transmitted packets, and end systems cannot determine who is responsible for dropping or delaying their traffic. As a result, they cannot verify that their ISPs are honoring their service level agreements, nor can they react to adverse network conditions appropriately. While current probing tools provide some assistance in this regard, they only give feedback on probes, not actual traffic. Moreover, service providers could, at any time, render their network opaque to such tools.**

We propose *AudIt*, an explicit *accountability interface*, through which ISPs can pro-actively supply feedback to traffic sources on loss and delay, at administrative-domain granularity. Notably, our interface is resistant to ISP lies and can be implemented with a modest NetFlow modification. On our Click-based prototype, playback of real traces from a Tier-1 ISP reveals less than 2% bandwidth overhead. Finally, our proposal benefits not only end systems, but also ISPs, who can now control the amount and quality of information revealed about their internals.

I. Introduction

The Internet is built around a best-effort service model that provides no guarantees ahead of time about when, or even if, packets will be delivered. Many have argued that this lack of guarantee played a key role in the Internet's success, enabling IP to run over a wide range of network technologies using simple and scalable algorithms. While some clamor for augmenting best-effort with quality-of-service assurances, few if any believe that the best-effort model should be discarded.

The Internet has also adopted the philosophy of not providing any after-the-fact information about the fate of packets. The rationale for not providing advance assurances—enabling flexibility of network technologies and simplicity of the forwarding path—does not apply to monitoring and reporting. The lack of such on-line monitoring tools (as opposed to probing tools, which we discuss later) more likely arises from strict adherence to layering and transparency according to which, from a host's perspective, all that matters is whether and when a packet was delivered, which can be determined by the endpoints themselves without help from the network. This line of thinking has resulted in an Internet that is transparent to success but opaque to failure.

It has long been a central Internet tenet that applications should adapt to current network conditions, but often the notion of adaptation was limited to purely end-to-end considerations such as congestion control and adaptive coding techniques. In such cases, not knowing where packet loss or delays occurred is no hindrance; all that matters is that they did occur and endpoint measurements are enough to establish that fact. However, there have been several recent efforts to extend adaptation to edge-controlled routing. Proposals, such as TRIAD [13], NIRA [30], and Platypus [26], allow end systems to control the domain-level path of their packets, while a more recent proposal [33] enables multi-path sources to choose failure-independent paths. To make an informed decision about such paths when current service is poor, an end system needs to know which domains are currently dropping or delaying its packets. We therefore contend that the Internet should not remain "opaque to failure" but should instead provide information about where packets are being dropped or delayed so that end systems (whether these be the source hosts themselves, or the originating domain) can intelligently adapt to current conditions.

There is also a simpler rationale for providing this information. Internet service is a contractual business; end users pay their ISP, and ISPs have either customer-provider or peering arrangements with each other. Providing some form of performance feedback would help establish whether providers (and peers) are adequately performing their duty. Laskowski and Chuang have showed that, without such accountability, optimal routes and innovation in the Internet are impossible [23].

Existing Internet probing tools such as ping and traceroute can help debug network problems. These tools are very effective in pinpointing long-lasting outages or persistent high-drop rates. However, because they only reveal how the network reacts to probe packets, not to previously sent packets, they fail to capture low-rate or sporadic misbehavior (e.g., an intermittent router failure, or malicious low-rate drop patterns [22]). As such, they have limited value when trying to make finely-tuned decisions about the reliability of a provider's service. More importantly, even if the Internet's behavior on probe traffic were enough to detect all problems, such probing tools reveal information at router granularity, at the border and within the interior of ISPs alike. One cannot expect that ISPs will remain so transparent to these tools: in what other industry do organizations allow free inspection of their internal infrastructure? We think it likely that the current trend of ISPs befuddling these tools will increase, preventing unauthorized probing.

Based on these considerations, we think the Internet would be well-served to move beyond the current situation, where there is no systematic way to learn the fate of packets, and all such performance monitoring relies on an ad hoc set of probing tools that provide more information than an ISP would like to reveal (its internal structure) and less information than an end system would like to know (what happened to the previously sent packets). We propose instead *AudIt*, an explicit *accountability interface*, through which ISPs report on their own performance. We argue that this is better than probing, both for the end systems and the ISPs: the former learn what happens to their traffic, not just their probes; the latter control what information they release regarding their business. We show that ISPs cannot misuse our interface to lie about their (or other ISPs') performance. We also present two case studies on implementing the interface to report on TCP traffic and demonstrate that it can be done with a modest NetFlow [1] modification and a reasonable amount of resources.

After a problem statement (§II) we define AudIt (§III) and describe how it can be used in the face of both honest and dishonest ISPs (§IV). Next, we present our case studies: first a straightforward implementation that provides accurate loss feedback (§V), then an extension that provides accurate delay feedback (§VI); we evaluate them in a software prototype (§VII). We close with a discussion of the bigger picture beyond what is covered in this paper (§VIII), related work (§IX) and our conclusions (§X).

II. Problem Setup

A. Goals

With this work, we wish to enable traffic sources to determine which administrative domains are losing and/or delaying their packets; an *administrative domain* (AD) is defined as a contiguous network administered by a single authority. Each administrative authority that provides accountability is free to choose how to present itself: an AD can correspond to a single Autonomous System (AS), a group of peering ASes, an entire ISP, or even a coalition of neighboring ISPs.

More specifically, our first goal is to provide a traffic source with enough feedback to determine: (1) a measure of how much of its outgoing traffic was dropped at which AD and (2) a measure of the delay experienced by its outgoing traffic through each traversed AD. The granularity of these "measures" can be very fine (e.g., per-packet metrics) or coarser (e.g., aggregate metrics over multiple packets).

It is critically important to ensure that the measures described above cannot be arbitrarily skewed by a malicious AD on the traffic path or off the path. Our second goal is to guarantee an upper bound on the error that a malicious AD can unobtrusively induce in each measure. We define a "malicious AD" in the threat model, below.

The flip side of the second goal is our third goal: when tampering of our monitoring metrics violates the error bound, this tampering should be attributable to a specific link between the tampering AD and its peer. In other words, egregious tampering should be *localizable*.

Our final goal is to "do no harm": our solution should not enable previously impossible attacks against innocent ADs; for example, we should not make denial-of-service attacks easier than they are now.

B. Threat Model

Our threat model allows an AD to be *benign*—that is, report what it measures dutifully—or *malicious*—that is, report inflated or deflated measurements for traffic traversing its infrastructure, including reporting having seen packets it did not, and reporting having not seen packets it did see. We place no restrictions on the ability of ADs to collude with other ADs (neighboring or otherwise).

$aggType$	Packet or TCP flow
$aggId$	Packet with digest D or TCP flow with specified {ToS, src IP/port, dst IP/port} tuple
$handoffPoint$	Inter-AD link #5 to AD X or all inter-AD links to AD X
$direction$	Incoming or outgoing
$numPkts$	10
$avgTime$	2007-08-08 18:02:49 and 454 msec CEST

TABLE I. Feedback entry fields and example contents.

Our threat model *does not* allow a malicious AD to modify or otherwise tamper with traffic reports from other ADs that it forwards. We justify this restriction by observing that, though ADs make no guarantees with regards to their own traffic (including reporting traffic they generate), they sign legally binding service-level agreements with their peers, which they would openly violate by manipulating neighbors' reporting packets; we believe that the majority of ISPs today would avoid such open violations. In Section VIII-A, we discuss expanding our threat model by removing this restriction, to address stronger adversaries, which we consider unrealistic for today's—but perhaps not tomorrow's—Internet.

III. Accountability Interface

In this section, we present an initial interface-level definition of the accountability facilities we propose. This is not intended as a rigorous mathematical background to feedback reporting and comprehension; rather, it is meant to illustrate the accountability facilities we advocate. In later sections we explain how a source AD can use the interface to determine the loss and delay of its own traffic.

A. AudIt Definition

A reporting AD organizes its feedback in *feedback packets*, each one including its identity and a set of *feedback entries*. At a high level, a feedback entry specifies a unidirectional *traffic aggregate*, a *hand-off point* where packets from this aggregate entered or exited the reporting AD, how many such packets were observed at this hand-off point, and *when* they were observed. This information is encoded in the feedback-entry fields stated in Table I; in the rest of this section, we discuss them in more detail.

The $aggType$ and $aggId$ fields together specify the traffic aggregate a feedback entry refers to. The interface allows for multiple aggregate types, so that each AD can choose its own granularity of reporting aggregates—as we discuss later, this choice affects the quality-overhead trade-off of the mechanism. Any rule that unambiguously specifies a set of packets sent by a source AD can be used to define an aggregate type; the only restriction is that aggregate types must be such that any two aggregates either have no packets in common or one is a subset of the other. For example, two aggregate types that honor this restriction are packets and TCP flows (as defined in §V-C).

The *handoffPoint* field describes a connection between the reporting AD and one of its peers, through which *aggId* packets transitioned from one AD to the other. It can specify one or more inter-AD links or the peer itself (implying that this connection consists of all inter-AD links with the specified peer). In this way, each AD can choose the level of detail at which it exposes its structure—for instance, by exposing one hand-off point per peer, an AD provides no information on the number of its inter-AD links. An AD makes publicly available the identities of its hand-off points, as well as the maximum acceptable delay across each hand-off point as agreed upon with the corresponding peer—"acceptable" in the sense that, if it is exceeded, the corresponding inter-AD links are considered to have failed.

The *direction* field specifies whether *aggId* packets entered or exited the reporting AD at *handoffPoint*. The *numPkts* field is a count of the *aggId* packets observed at *handoffPoint*, while *avgTime* is a timestamp that corresponds to the (absolute) average time at which these packets were observed.

B. An Informal Aggregate and Feedback Algebra

To express the relationship between two aggregates α and β of traffic originating at certain source AD, we use set notation. Any two such aggregates are *combinable* iff

- $\alpha \subseteq \beta$, all of α's packets also belong to β, or
- $\beta \subseteq \alpha$, all of β's packets also belong to α, or
- $\alpha \cap \beta = \emptyset$, α and β have no packets in common.

To express the combination of two or more combinable aggregates, we use the union operator. For example, if $\alpha \subseteq \beta$, then $\alpha \cup \beta = \beta$. As with set union, combination is associative and commutative (e.g., $\cup_{\forall i}\{\alpha_i\} = \alpha_k \cup (\cup_{\forall i, i\neq k}\{\alpha_i\})$).

To denote a particular feedback entry, we use vector notation, e.g., $\tilde{x}$. To denote a particular field within feedback entry $\tilde{x}$, we use notation $\tilde{x}.\langle \textit{field name} \rangle$, e.g., $\tilde{x}.aggId$. As a convention, we use the same symbol to denote an aggregate and its identifier. Feedback entries from the same AD and with the same direction can be combined (using the combinator $+$) to form feedback entries for the combined aggregates. For instance, if a given AD's feedback entries $\tilde{x}$ and $\tilde{y}$ refer to aggregates α and β, respectively, then the feedback entry $\tilde{x}+\tilde{y}$ refers to the aggregate $\alpha \cup \beta$. Recall that not all aggregates can be combined. Table II defines the combinator $+$.

IV. Using AudIt

In this section we present how AudIt can be used to provide traffic sources with performance feedback. We first describe how sources can decipher reports from honest ISPs, then how dishonest reports can be detected via feedback inconsistencies, leading to lie localization.

A. Honest Reporters

In the absence of dishonest feedback reporters, a source can combine a collection of feedback entries on aggregate α from multiple ADs to determine α's AD-level path. It can also combine the packet counts and average timestamps collected at all the entrances and exits of each reporting AD to compute the number of α packets dropped or the delay incurred by α packets per AD along that path.

$\tilde{x}.aggId$	$=$	$\cup_{\forall i}\{\tilde{x}_i.aggId\}$
$\tilde{x}.handoffPoint$	$=$	$\cup_{\forall i}\{\tilde{x}_i.handoffPoint\}$
$\tilde{x}.direction$	$=$	$\tilde{x}_i.direction$
$\tilde{x}.numPkts$	$=$	$\Sigma_{\forall i}\{\tilde{x}_i.numPkts\}$
$\tilde{x}.avgTime$	$=$	$\frac{\Sigma_{\forall i}\{\tilde{x}_i.numPkts \cdot \tilde{x}_i.avgTime\}}{\tilde{x}.numPkts}$

TABLE II. Definition of $\tilde{x} = +\tilde{x}_i$, when all feedback entries $\tilde{x}_i$ are produced by the same AD, they all have the same direction $\tilde{x}_i.direction$, and all aggregates $\tilde{x}_i.aggId$ are combinable.

More specifically, if AD X produced feedback entry $\tilde{x}$ on aggregate α, and AD Y produced feedback entry $\tilde{y}$, the source can determine that X delivered α packets to Y, if $\tilde{x}.handoffPoint = \tilde{y}.handoffPoint$, $\tilde{x}.direction = out$, and $\tilde{y}.direction = in$. Given all feedback entries $\tilde{x_i}$ produced by AD X on α, the source can determine the following:

1) The number of α packets lost within X is $L = \tilde{x}_{out}.numPkts - \tilde{x}_{in}.numPkts$, where $\tilde{x}_{in} = \Sigma_{\{i, \tilde{x}_i.direction=in\}}\{\tilde{x}_i\}$ and $\tilde{x}_{out} = \Sigma_{\{i, \tilde{x}_i.direction=out\}}\{\tilde{x}_i\}$.
2) The average delay incurred by α packets within X is $\tilde{x}_{out}.avgTime - \tilde{x}_{in}.avgTime$, if $L = 0$.[1]

Since different ADs may report on different aggregate types, it is up to the source to do the necessary combinations (by applying the simple algebra of §III-B). We illustrate with a simple example. Suppose an aggregate that consists of three packets crosses a hand-off point from AD X to AD Y. In response, X produces one feedback entry $\tilde{x}$ on the entire aggregate, whereas Y produces three feedback entries $\tilde{y}_1$, $\tilde{y}_2$, and $\tilde{y}_3$, one for each packet. It is up to the source to determine that $\tilde{y}_1$, $\tilde{y}_2$, and $\tilde{y}_3$ are combinable, and that $\tilde{x} = \tilde{y}_1 + \tilde{y}_2 + \tilde{y}_3$. Then, the source can order the reports to determine the aggregate's path, as well as individual packet loss and delay measures on each AD.

B. On-path Lies

We now turn to the case in which an AD misrepresents its performance when reporting on a particular aggregate. We seek to answer two questions: when can a source detect such lies and, when it does, can it identify the liars?

1) Detection: We start with the observation that correct feedback entries from two peering ADs on the same traffic aggregate satisfy certain consistency conditions, as long as the inter-AD links between the two ADs do not drop, reorder, or inconsistently delay packets. If the two ADs' feedback entries on the same aggregate disagree on (1) how many packets the earlier AD delivered to the later AD, or (2) when, on average, it delivered them, then either one of them is lying, or there is a problem with the inter-AD link between them.

Definition: Consider a traffic aggregate α that crosses a hand-off point from AD X to AD Y; the two ADs produce feedback entries on α, denoted by $\tilde{x}$ and $\tilde{y}$, respectively, with

[1]This does not mean that an AD cannot provide delay feedback on traffic that incurs loss; we show how to do that in §VI.

$\tilde{x}.handoffPoint = \tilde{y}.handoffPoint$. Feedback entries $\tilde{x}$ and $\tilde{y}$ are *consistent* with each other, iff

- $\tilde{y}.numPkts = \tilde{x}.numPkts$
- $\tilde{y}.avgTime - \tilde{x}.avgTime \leq \tau$, where τ is the maximum acceptable one-way delay across $\tilde{x}.handoffPoint$.

A second observation is that, when feedback entries on the same aggregate are consistent, an involved AD can only lie about its performance by implicating one or more of its peers. We illustrate with two examples.

Consider again traffic aggregate α and ADs X and Y from the definition above. Suppose Y drops one of the packets, but, instead of admitting the loss, it claims it never got the lost packet in the first place, i.e., it reports receiving from X one fewer packet than it actually did (or $\tilde{y}.numPkts = \tilde{x}.numPkts - 1$); this implies that either X did not deliver to Y all α packets that it reported, or the inter-AD link between X and Y is lossy.

Now suppose Y tries to hide some of the delay incurred by α in its network, by claiming that, on average, it received α packets 10 msec later than it actually did (or $\tilde{y}.avgTime - \tilde{x}.avgTime = \tau + 10$ msec). This necessarily implies that either X delivered the packets at that time, or the inter-AD link between X and Y introduced an additional 10-msec delay.

In both examples, Y is essentially blaming its own loss or delay on X; alternatively, it could blame them on the next AD on α's path.

Definition: Consider a traffic aggregate α that crosses a hand-off point from AD X to AD Y; the two ADs produce feedback entries on α, denoted by $\tilde{x}$ and $\tilde{y}$, respectively, with $\tilde{x}.handoffPoint = \tilde{y}.handoffPoint$. Suppose X delivers through this hand-off point p packets of aggregate α to Y at average time t. We say that:

- "Y blames loss δp on X" with respect to α, if $\tilde{y}.numPkts = p - \delta p$.
- "Y blames average delay δt on X" with respect to α, if $\tilde{y}.avgTime = t + \tau + \delta t$.
- "X blames loss δp on Y" with respect to α, if $\tilde{x}.numPkts = p + \delta p$.
- "X blames average delay δt on Y" with respect to α, if $\tilde{x}.avgTime = t - \tau - \delta t$.

Finally, we observe that, if an AD blames loss and/or average delay on one of its peers, unless the peer is in on the lie, the lie is bound to result in inconsistent feedback entries between the liar and the peer—for instance, X reports delivering p α packets to Y, while Y reports receiving $p - \delta p$ α packets from X. Inconsistency alerts the receiver of the corresponding feedback entries to the fact that something is wrong (either an AD is lying or an inter-AD link is problematic), triggering further investigation. So, one measure of the harm a lying AD can do is the extent to which it can blame an innocent peer without causing any feedback inconsistencies.

Lemma 4.1: *If AD X produces correct feedback entries on traffic aggregate α, then none of X's peers can blame any loss or average delay on X with respect to α without causing a feedback inconsistency* (we omit the straightforward proof for lack of space).

Interestingly, the converse is also true: when AD X does *not* report on α traffic at all, then its peers can cause X to be reported incompetent with respect to α. This is one of the basic incentives for deploying our accountability interface: the more information an AD generates about its performance, the more difficult it is for its peers to undetectably blame their faults on it.

Note that Lemma 4.1 holds even when peering ADs use different aggregate types. For instance, consider traffic aggregate α that crosses a hand-off point from AD X to AD L (L for "liar"). Suppose X produces a single feedback entry on α, whereas L produces one feedback entry for each α packet. Now L can lie about which individual α packets X delivered, but it must still ensure that the total number of received packets it reports is equal to the number of delivered packets reported by X. Similarly, L can lie about the time it received each individual packet, but it must still choose the reported entry times such that their average is equal to the average exit time reported by X. Essentially, L can wrongly accuse X of losing or delaying an individual α packet (and X cannot dispute the claim, because it is not reporting on each packet), but it cannot blame any loss or average delay on X with respect to α without causing feedback inconsistencies.

Between successive ADs, feedback inconsistencies are inescapable at the granularity of the *nearest common superset* of the aggregates reported for the same traffic. Being able to find that nearest common superset efficiently (that is, without combinatorial searches over all feedback entries received at the source) is an essential criterion determining which aggregate types are compatible with AudIt.

2) Localization: When source S receives inconsistent feedback entries from a pair of ADs X and Y, either one of them is lying about its performance, or at least one inter-AD link between them is faulty. The source cannot determine which of these are true, but it can narrow down the problem to the X-Y pair. This may be useful to the source (e.g., if the source is connected though multiple ISPs, it may be able to route its traffic avoiding the suspicious X-Y link altogether), but it is not enough for accountability: if ADs can lie about their performance and then point fingers at their peers, there is no incentive to tell the truth.

We address this problem by exposing lying ADs to the peers they implicated. Continuing with the above example, if source S receives inconsistent feedback entries from X and Y, it subsequently asks X and Y for a signed version of these entries. If an AD responds with a signed entry that differs from the original (unsigned) one, then S concludes that AD is lying (within our threat model). If both ADs insist on their original reports, S sends both signed entries to both X and Y. From that point on, it is up to the two peers to sort out their differences: if both ADs insist they are telling the truth, they can investigate their inter-AD link; if no problems are found with the link, i.e., the inconsistency was due to a lie, then the lying AD is exposed to the peer it implicated.

AudIt does not mandate how peering ADs investigate and resolve their disputes over feedback inconsistencies—that depends on the debugging tools they have at their disposal as much as their business relationship. Whatever the process, it provides a strong incentive for ADs to be honest: if lying means implicating a peer, who will deterministically learn that it has been implicated, then lying means entering a (potentially legal) dispute with that peer and damaging the corresponding business relationship. Given the nature of today's ISP business, in which peers sign either provider-customer SLAs or peering agreements, we believe that an ISP would not risk losing a peer's trust.

3) The Role of Inter-AD Links: One could argue that feedback inconsistencies between two peering ADs are impossible to properly ascribe when in fact it is the inter-AD link between them that has failed. In practice, an innocent AD should be able to discover the truth: An inter-AD link can be a physical link connected at each end to elements belonging to the two ADs; the only way for such a link to introduce loss or unpredictable delay is for it to be physically damaged, which is straightforward to debug with the right equipment. Alternatively, an inter-AD link can consist of two physical links plugged into a switch located at an Internet exchange point; in this case, investigating an inconsistency involves verifying the health of the physical links as well as the loss and delays introduced by the switch.

Of course, we cannot preclude the case where an inter-AD link goes through a sophisticated exchange point that introduces multiple active elements in the datapath. In that case, however, the exchange point itself becomes an administrative entity that receives and delivers packets, which means that it should also support AudIt, otherwise ADs will be free to blame their faults on it. In general, the idea is that hard-to-debug entities export the accountability interface, so that faults can be tracked down to a pair of such entities and an easy-to-debug element between them, like a physical link; then, when two entities send inconsistent feedback, it is easy for an innocent entity to determine whether the other one is lying or the element between them has failed.

C. Off-path Lies

Besides lying about its forwarding performance, a malicious AD may also choose to lie about having seen an aggregate when it, in fact, has not. We refer to this misbehavior as "off-path lying." An adversary's impersonating a legitimate on-path AD (i.e., *forging* feedback entries for another AD) is an authentication issue that is handled in an implementation-specific fashion (see §V-E).

When all ADs on an aggregate's path provide feedback, the source will not be tempted to consider feedback from an AD L situated off the actual path of the aggregate: if no AD downstream of the source designates L as the next hop, L cannot present itself as "on path." An on-path AD M that misrepresents its next hop to be L is itself malicious and can be caught by the same feedback inconsistencies described in the previous section: M's downstream AD would report M as the previous hop for the aggregate in conflict to M's reporting of L as its next hop for the aggregate.

In general, an off-path AD L cannot blame loss or delay with respect to aggregate α on an innocent AD any more than an on-path AD can—i.e., not at all, as long as the innocent AD produces correct feedback on α. In a scenario where not all ADs on α's path provide timely feedback, and, moreover, the source does not know α's AD-level path, L may be able to produce credible feedback on α and present itself as being on path; however, it cannot blame any loss or delay with respect to α on any AD correctly reporting on α. Note that, to produce credible feedback, L needs help by an on-path colluder; without it, L would be hard pressed to pick the right number of packets, a consistent average entry time, etc., at the risk of being identified as a generator of false feedback and penalized in its business with its partners.

V. Basic Feedback on TCP Traffic

We now present our first case study: how an ISP can implement AudIt for reporting TCP-flow statistics, in particular, the number of packets lost and, if that is zero, the average delay incurred by each TCP flow within its network.

A. Checkpoints

Statistics are collected at designated *checkpoints*, located on inter-AD links. Physically, a checkpoint can be a monitoring module running inside a border router, or a separate box positioned to passively tap the link. Conceptually, it consists of a link tap, a clock, short- and long-term state and a sending buffer.

An AD places checkpoints on all the links through which traffic enters and exits its network. Each checkpoint typically plays two roles: as an entry point, it collects statistics on traffic entering the AD; as an exit point, it collects statistics on traffic exiting the AD.

Each AD must keep its checkpoint clocks roughly synchronized. One option is to use NTP and get an accuracy of a few hundred microseconds (as long as each checkpoint is located in the same local network with an NTP server) [11]; a better one is to equip each checkpoint with a GPS receiver (currently costing about $200) and get an accuracy of 340 nsec [4]. Checkpoints from different ADs located on the same inter-AD link need only keep track of their clock drift.

B. The Accountability Center

Each AD maintains an *accountability center* as part of its network management platform. Physically, this can be a module running inside a management node, or a cluster of nodes, depending on the size of the AD and the amount of traffic it generates and forwards.

An accountability center exports two interfaces. As part of a source AD S, it exports a *feedback receiver* interface, where reporting ADs can send their feedback regarding traffic generated by S; the address of this interface is publicly available through DNS. As part of a reporting AD X, it exports a *follow up* interface (e.g., over HTTPS), where feedback receivers can request/provide signed statements in case of

Field name	**Description**	**# bits**
$tcpId$	ToS, src IP/port, dst IP/port	104
$numPkts$	Number of packets with $tcpId$ observed at this checkpoint	8
$firstArrival$	Time the first packet was observed at this checkpoint	32
$lastArrival$	Time the last packet was observed at this checkpoint	32
$avgTime$	Average time at which the packets were observed	32
$closed$	Is this piece of state "closed"?	1

TABLE III. Short-term state maintained per TCP flow at each checkpoint. All the timestamps are in milliseconds.

feedback inconsistencies (see §IV-B.2), or verify that a certain IP address corresponds to a checkpoint from X.

C. Packet Classification per TCP Flow and Short-term State

A checkpoint considers a sequence of packets to belong to the same aggregate of type "TCP flow," when both of the following conditions are true:

- all packets have the same {ToS, src IP/port, dst IP/port} tuple, and
- any FIN or RST packet is the last one.

Each checkpoint maintains short-term state per TCP flow, which is organized in *flow records* (see Table III). The only difference between a NetFlow cache and a checkpoint's short-term state is that the latter includes the average time at which each flow's packets were observed.

A record is "closed," i.e., stops getting updated, for any of the following reasons: (i) The flow ended, i.e., a FIN or RST packet from that flow was observed. (ii) Inactivity, i.e., $currentTime - lastArrival > maxIdle$; in our implementation, we use $maxIdle = 15$ sec, i.e., the NetFlow default for maximum packet inter-arrival time within a flow. (iii) Age, i.e., $currentTime - firstArrival > maxAge$; we use $maxAge = 60$ sec (less than the 30-minute NetFlow default), because we want to collect aggregate statistics on a long flow at least every minute. (iv) The number of packets exceeded the corresponding field size ($numPkts > 255$).

D. Determining Where to Report

To send collected feedback to the corresponding source ADs, the accountability center of each reporting AD builds a *feedback-receiver map*, which maps IP prefixes to their origin ADs and the corresponding feedback-receiver addresses; this table is then distributed to all the checkpoints of the AD. The accountability center builds the feedback-receiver map in two stages: first, it compiles an IP-prefix-to-origin-AD map using BGP data from the AD's border routers; then it looks up the feedback-receiver address for each origin AD and adds that information to the map. Given that IP-prefix-to-AD maps computed from BGP tables are known to be ambiguous [25], we explain next why these limitations have no impact on our mechanism.

In the past, IP-prefix-to-AD maps have been compiled in the context of AS-level traceroute [25], i.e., when trying to map the source addresses of ICMP TIME_EXCEEDED messages to ASes. Mapping those addresses (which belong to IP-router interfaces) can be tricky in several scenarios. First, ASes do not always advertise the addresses of their router interfaces. Second, some router interfaces are physically located at exchange points and may be advertised by more than one AS. Third, if a non-BGP speaking network has multiple providers, each one of them advertises the network's prefixes, which, as a result, appear to belong to multiple ASes. The first two scenarios are not relevant to our mechanism, because we only need to map addresses of TCP sources, never private infrastructure. The third scenario is straightforward to handle: multi-homed, non-BGP speaking networks, either do not receive feedback or receive feedback through their BGP speaking providers.

E. Long-term State and Statistics Reporting

Each checkpoint reads its short-term state every T_s seconds, creates feedback entries from its closed records, packages them per source AD (using the feedback-receiver map), copies them to local storage (from where they expire after T_l hours), and sends them to the corresponding feedback receiver via UDP. Source ADs that do not advertise a feedback-receiver address do not get any feedback on their traffic. Feedback packets lost due to congestion can be recovered through the follow-up channel (see §V-F).

To identify and drop spoofed feedback packets, a feedback receiver uses a lightweight authentication scheme, reminiscent of SYN-cookies [10]. When first contacted by a checkpoint, the feedback receiver verifies that the sender's address indeed corresponds to a checkpoint; then it responds with a random nonce, which it stores locally. All subsequent reports from that checkpoint to the feedback receiver carry increments of that nonce, much like a TCP sequence number. The feedback receiver periodically changes the nonce for each checkpoint and establishes it with a new handshake.

When a flow record is closed due to inactivity, age, or overflow of the $numPkts$ field, the corresponding feedback entry can temporarily lead the source AD to wrong conclusions. For example, if a transit AD delays a packet by more than $maxIdle$, the exit point will close the corresponding flow record before observing the delayed packet, potentially "breaking" the flow into two separate feedback entries; after receiving the first feedback entry, the source AD may falsely conclude that the reporting AD lost the second part of the flow. Such errors are corrected once the corresponding TCP flow has ended, and the source AD has collected all related feedback entries.

F. Follow-up Channel

If a feedback receiver is missing expected feedback or identifies a feedback inconsistency, it uses the follow-up interfaces of the involved ADs to resolve the issue as described in §IV-B.2. Upon receiving a request, the accountability center retrieves the relevant information from the corresponding

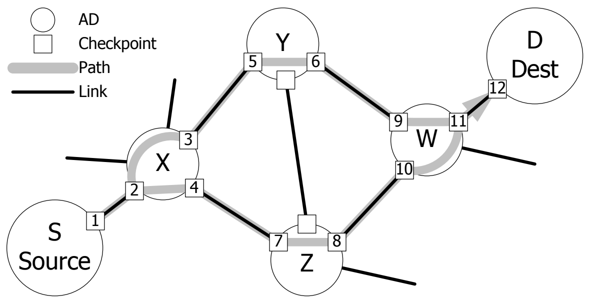

Fig. 1. AD-granularity view of the paths taken by a source's aggregate to a remote destination.

checkpoints and puts together the requested statement. Feedback receivers do not directly contact checkpoints.

G. Limitations

The benefit of the implementation presented in this section is simplicity: each checkpoint collects NetFlow-style statistics and sends them to the corresponding source ADs. There are two cases, however, in which providing per-TCP-flow statistics is insufficient to characterize AD performance with respect to TCP traffic; we discuss these cases here, before addressing them in the next section.

The first case is split TCP flows. Consider the following scenario: In Figure 1, traffic from S to D is normally routed through checkpoints 3, 5, 6, 9 and 11; the typical delay between checkpoints 9 and 11 is 10 msec. Then X malfunctions (e.g., a link goes down) and routes two packets from the same TCP flow through different paths; as a result, W receives the two packets through different entry points. The first packet follows the normal path (and incurs the typical 10-msec delay), but the second one follows a longer path and incurs a 100-ms delay within W's network. In this scenario, AD W introduces higher average delay than normal into a TCP flow as a result of a malfunction in a previous AD. Yet, if W collects per-TCP-flow statistics (i.e., exports the average entry and exit time for the two packets), a source receiving its feedback can only determine that the average delay across W was 55 msec and potentially conclude that W's performance decreased.

The other case is long, delay-sensitive TCP flows that incur packet loss. It is possible that the source of such a flow would want to know the average delay incurred within each AD *by the packets that were successfully delivered by that AD*. Reporting the average time at which the flow's packets entered each AD is not enough to compute this information, because that corresponds to *all* the packets that entered the AD, not the ones that made it to the exit.

VI. Accurate Delay Feedback on TCP Traffic

We now show how an ISP can extend the implementation presented in §V to accurately report on the average delay incurred in its network by split TCP flows and/or flows that incur packet loss.

This case study concerns ADs whose internal paths do not reorder same-class packets, i.e., if two packets from the same class of service follow the same sequence of router interfaces within the AD, they are guaranteed to enter and exit the AD in the same order. Although the IP protocol itself does not provide this guarantee, to the best of our knowledge, modern routers (and the providers that use them) generally do.[2]

Field name	Description	# bits
$entryPoint$	Identity of entry checkpoint	16
$entryTime$	Average entry time of the packets observed at this exit point	32

TABLE IV. Short-term state maintained at each exit point per single-path TCP flow. The entry time is in milliseconds.

A. Packet Classification per Single-path TCP Flow

A checkpoint considers a sequence of packets to belong to the same aggregate of type "single-path TCP flow," when these packets

- belong to the same TCP flow (as defined in §V-C) and
- cross the same entry and exit checkpoints of this AD.

Note that collecting packet counts and average timestamps at the granularity of single-path TCP flows overcomes the limitations described in §V-G: First, using these statistics, a source AD can compute the average delay incurred by a split flow across each individual intra-AD path of the reporting AD. Second, since packets are assigned to flows based on their entry and exit checkpoints, the average entry and exit timestamps reported by an AD on a certain flow always refer to the same set of packets, which allows a source AD to accurately compute the average delay incurred by these packets within the reporting AD.

B. Short-term State

Checkpoints collect different types of statistics depending on their role: entry points maintain state per TCP flow, whereas exit points maintain state per single-path TCP flow. The latter is also organized in records, each one including the fields of Table III plus two extra fields summarized in Table IV: an $entryPoint$ field (which specifies the entry point for this flow) and an $entryTime$ field (which specifies the average time at which this aggregate's packets entered this AD).

It may seem counter-intuitive, at first, that we maintain the average *entry* time of an aggregate at the corresponding *exit* point. The reason is that an entry point cannot assign packets to single-path TCP flows (and compute the corresponding average timestamps), because it cannot know *if* and *where* each observed packet will exit the AD.

C. Statistics Collection

We have established that the right place to assign packets to a single-path TCP flow and compute their average entry time is the exit point that observes these packets. This creates an implementation challenge: an exit point must be able to

[2]An exception is Juniper's M160 OC192 linecard, which introduced packet reordering due to its parallel structure [6]. Reordering was eliminated in the company's next core router [7] after bad publicity—notably a comparison test with Cisco's highest-end, at the time, router, showing that the latter introduced no reordering [5].

determine *where* and *when* each observed packet entered the AD. One would think that at least the "where" question could be answered based on routing state; unfortunately, interior and external routing tables are generally insufficient [18].

A seemingly straightforward solution is packet annotation: When a checkpoint observes a packet entering its AD, it reads the packet's TCP/IP headers, updates its (per TCP flow) state accordingly, and annotates the packet with its IP address and the current time; the checkpoint that observes the packet exiting the AD uses the annotation to compute the $entryPoint$ and $entryTime$ fields and update its (per single-path TCP flow) state accordingly. Albeit conceptually simple, this approach would face deployment issues: even though packet size and content modification at line speed is within the capabilities of modern hardware, ISPs are typically not equipped to perform it, especially at the Internet core.

Fortunately, in a typical modern AD, it is feasible to "emulate" packet annotation: Inter-AD routes do not change all that frequently, which allows an exit point to correctly guess with a high probability the entry point of an observed packet. Moreover, modern routers do not arbitrarily reorder packets, which allows an exit point to bound the delay incurred by a packet without knowing its exact entry time. Based on these observations, we propose a solution, where the exit point is not explicitly told the entry point and time of each observed packet, but rather performs *informed guesses* and fixes any mistakes after the fact.

1) Entry-point Disambiguation: Each entry point processes the source and destination addresses of observed packets and builds a history of source-destination prefix pairs. All entry points of a given AD periodically send updates of their history to the AD's accountability center, which uses them to build an *entry-point map* from source-destination prefix pairs to candidate entry points. Updates of this map are periodically distributed to all exit points of the AD, which use it to determine potential entry points for observed flows.

2) Entry-time Disambiguation: Moreover, each entry point sends to each exit point a *marker*, i.e., a control packet that includes a timestamp corresponding to its birth time, every T_m time units. Whenever an exit point observes a non-marker packet, it can determine a lower bound on the packet's entry time, by assuming the packet entered the AD right after the last marker from the corresponding entry point. For instance, suppose checkpoint 6, in Figure 1, observes a packet exiting AD Y and guesses that this packet entered at checkpoint 5; if the last marker from checkpoint 5 bore timestamp t, checkpoint 6 concludes that the packet cannot have entered Y before time t. For lack of space, we omit the details of handling routing changes, multiple intra-AD paths and multiple service classes.

3) Short-term State Update: Whenever an exit point observes a new TCP flow, it first uses the entry-point map to guess a set of candidate entry points, then creates one flow record for each one of them. Upon observing subsequent packets from the same TCP flow, the exit point updates the $numPkts$, $firstArrival$, $lastArrival$, $avgTime$, and $entryTime$ fields of all the corresponding records. To update the $entryTime$ field, the exit point assumes that the packet entered right after the last marker from the corresponding $entryPoint$. When there are no routing changes and no $entryPoint$ errors (see next paragraph), this overestimates the average delay incurred by a flow at most by T_m.

4) Error Correction: It is possible that the version of the entry-point map at a certain exit point is temporarily outdated: an entry point has observed a new {source prefix, destination prefix} pair, but the corresponding update has not reached the accountability center or the exit point yet. This may result in the exit point observing a new TCP flow and failing to create a flow record that corresponds to its actual entry point. To deal with this case, each exit point remembers the prefix pairs it looked up recently and the flow records it created; if it receives an update of the entry-point map that concerns one of these prefix pairs, it creates a new flow record according to the new mapping, assuming that all packets of the flow incurred the maximum possible delay.

D. Statistics Reporting

Each exit point reads its short-term state every T_s seconds, organizes closed flow records per $entryPoint$, and sends them to the corresponding entry point. Whenever an entry point receives a record from an exit point, it looks up the corresponding $tcpId$ in its local state and associates the record with the matching local one; if no match is found, the record is discarded. Eventually, an entry point associates each local TCP flow record with one or more matching single-path TCP flow records sent by exit points, produces a set of "coalesced" feedback entries (avoiding to repeat the 104-bit long $tcpId$ with every entry), copies them to local storage, and sends them to the corresponding source AD. To compute the number of packets from each TCP flow that entered/exited the reporting AD and the corresponding average timestamps, the source AD must combine all feedback entries on single-path TCP flows with the same $tcpId$ and $direction$ sent by the reporting AD.

We illustrate with an example. In Figure 1, source AD S sends out TCP flow f, which consists of 3 packets. The three packets enter X at times 9, 10, and 11. X loses the first one, delivers the second one to Y at time 20, and the third one to Z at time 26. The first packet enters W at time 50 and is delivered to D at time 60, while the second one enters W at time 55 and is delivered at time 155. Table V describes the content of the (honest) feedback entries produced by X and W on f. To compute the number of f packets that exited X and their average exit time, S combines the feedback entries produced by checkpoints 3 and 4 (3rd and 5th line in Table V). Similarly, to compute the number of f packets that entered W and their average entry time, S combines the feedback entries produced by checkpoints 9 and 10 (6th and 9th line in Table V).

E. Limitations

In the beginning of this section, we set out to produce accurate delay statistics for TCP flows that are split across multiple paths and/or incur loss. We mentioned packet annotation as a

	aggType	*aggId*	*handoff Point*	*dir*	*num Pkts*	*avg Time*
1	TCP	f	1-2	*in*	3	10
2	SP TCP	f, 2-3	1-2	*in*	1	10
3	SP TCP	f, 2-3	3-5	*out*	1	20
4	SP TCP	f, 2-4	1-2	*in*	1	11
5	SP TCP	f, 2-4	4-7	*out*	1	26
6	TCP	f	6-9	*in*	1	50
7	SP TCP	f, 9-11	6-9	*in*	2	98
8	SP TCP	f, 9-11	11-12	*out*	2	108
9	TCP	f	8-10	*in*	1	55
10	SP TCP	f, 10-11	8-10	*in*	2	8
11	SP TCP	f, 10-11	11-12	*out*	2	108

TABLE V. Feedback sent by ADs X (top half) and W (bottom half) in Figure 1 to source AD S regarding TCP flow f. "SP TCP" stands for "single-path TCP flow." The format of the $aggId$ field depends on $aggType$: for TCP flows, it consists of $tcpId$; for single-path TCP flows, it consists of $tcpId$ and an $entryPoint$-$exitPoint$ pair.

conceptually straightforward but expensive solution; then we "approximated" that solution with a cheaper one at the cost of introducing certain inaccuracies in the collected statistics. We now discuss these inaccuracies and how they qualitatively affect the provided feedback.

When a flow enters an AD through multiple entry points, the corresponding exit points have no way of guessing which packet entered at which entry point; the best they can do is assume that the entire flow entered at each of the candidate entry points and compute the corresponding average entry times. This may not reveal the exact delay incurred by the flow along each path, but does provide information on the performance of each path that carried the flow. For instance, consider again the scenario depicted in Figure 1: Checkpoint 11 guesses that flow f entered through checkpoints 9 and/or 10, creates two flow records (one for each entry point), and updates both of them with every observed f packet. The resulting feedback entries (see Table V, lines 7 and 8 for the path between checkpoints 9 and 11, and lines 10 and 11 for the path between checkpoints 10 and 11) allow S to estimate the performance of the two W paths—specifically, what the average delay incurred by f within W would have been, if both f packets had entered W through checkpoint 9 or 10.

Marker-based estimation of a packet's entry time relies on the assumption that same-class packets are not reordered along a single intra-AD path; a router malfunction that causes reordering can also cause an exit point to produce wrong entry-time estimates. Although, in practice, there are ways to alleviate the effects of such malfunctions, there is no clean way to provably bound the error they can introduce—at least not without assuming a maximum intra-AD delay. Both this and the previous limitation are due to the use of markers and can be avoided at the cost of using packet annotation.

We close with the observation that Lemma 4.1 does not apply to single-path TCP flows: unlike a TCP flow, a single-path TCP flow is unique to the AD that produced it, because it consists of the packets that were successfully delivered from a specific entry point to a specific exit point of that AD; hence, no two ADs can produce feedback on (and be inconsistent with each other with respect to) such a flow. For instance, W cannot lie about f's average exit time from its network (because that would cause an inconsistency with f's average entry time in D), but it *can* lie about the exit time of each of the two f packets that it successfully delivered (as long as D does not report the entry time of each individual f packet). In general, the extent to which an AD can lie depends on the detail at which its peers report their performance; if all peers report on TCP flows, an AD is free to report any performance it wants for its internal paths, as long as the average per-TCP-flow performance across all paths matches the peers' reports.

VII. Overhead Evaluation

We now evaluate the implementation proposed in §VI, based on a software prototype and real traces from OC-48 links of a Tier-1 ISP (obtained from CAIDA [3]).

A. Processing and Memory Overhead

We implemented a checkpoint prototype using the Click modular router [21]. Our prototype consists of a NetFlow-like traffic monitoring module and an accountability module. The former observes forwarded packets and collects per-TCP-flow state (as an entry point) and per-single-path-TCP-flow state (as an exit point). The accountability module periodically reads the collected statistics, packs closed flows into feedback packets, and sends them to the corresponding entry points (as an exit point) or source ADs (as an entry point); it is also responsible for sending and processing marker packets. We deployed this prototype on two PCs, each with a Xeon 3.8 GHz processor and 4 Gbytes of memory. We set up a simple testbed, where one PC acted as an entry point and the other as an exit point.

The goal of the experiment was to evaluate the performance of the accountability module—in an actual checkpoint implementation, the traffic-monitoring module would be implemented in hardware, close to the data path, e.g., as a NetFlow engine. More specifically, our goal was to test whether an off-the-shelf processor with a credible amount of memory can process per-flow statistics collected at a high-speed link and generate the corresponding feedback in real time. To this end, we emulated an OC192 link to our entry and from our exit point, i.e., we caused the traffic-monitoring modules running on the two PCs to generate $250,000$ new flow records per second—assuming $5,000$ bytes per flow, this corresponds roughly to 10 Gbps; each flow lasted for 20 seconds, leading to a total of 5 million concurrent flows. Moreover, we emulated a 100-checkpoint topology, i.e., the exit point thought it was sending its feedback to 99 entry points, whereas the entry point thought it was collecting its feedback from 99 exit points. We used $T_s = 10$ sec, $T_l = 5$ h, and $T_m = 5$ msec. Our accountability modules successfully sustained this flow rate; they started falling behind under a load of $500,000$ new flows per second.

As far as state is concerned, each checkpoint maintains two types: the short-term state described in Tables III and IV (38 bytes per single-path TCP flow) and a history of the statistics it has produced within the last T_l hours. With 1 GB of memory

Trace rate (Mbps)	386	833	330	294	115	145
Avg flow size (KB)	10.9	9.8	11.8	9	6.5	5.2
BW overhead (%)	0.8	0.9	0.8	1	1.4	1.8

TABLE VI. Trace characteristics (rate and average flow size) and bandwidth overhead. $T_s = 10$ sec and $T_l = 5$ h.

and 200 GB of storage, a checkpoint could handle roughly 20 million concurrent TCP flows and keep a 5-hour history on a billion flows per hour.

B. Bandwidth Overhead

Our implementation introduces three types of bandwidth overhead: the overhead due to marker packets incurred by each reporting AD, the (also intra-AD) overhead of sending feedback from exit to entry points, and that of receiving feedback from multiple ADs incurred by each source AD. We examine each one below.

The intra-AD overhead introduced by markers is independent of the amount of forwarded traffic: 64 bytes (the minimum packet size) every T_m time units for every entry-exit point pair that exchanges traffic. Each AD can use T_m as a knob to determine the balance between overhead and feedback quality. For instance, to achieve delay accuracy $T_m = 5$ msec, each pair of entry-exit points must exchange 100 Kbps of marker traffic; for an AD with 100 inter-AD links, this corresponds to a total of 10 Mbps of marker traffic per checkpoint (in each direction).[3]

The intra-AD overhead introduced by feedback flowing from exit to entry points depends on (1) the average TCP flow size and (2) the size of the flow records sent by exit points, which is 18 bytes in our implementation (we omit the formatting details for lack of space). To get an estimate of the average TCP flow size, we looked at six traces from a Tier 1 ISP, provided by CAIDA (see Table VI); we chose $5,000$ bytes per flow as a representative number, as all our traces showed a higher average flow size, which would reduce the overhead thanks to amortization. Assuming this average flow size and a single exit point per TCP flow as the common case, feedback introduces roughly 0.36% bandwidth overhead per entry point, where the percentage is computed over the throughput of the traffic observed by the entry point.

Similarly, the overhead incurred by a source AD depends on the average TCP flow size and the size of the feedback entries sent by each AD (in our implementation, 23 bytes per single-path TCP flow), but also the average number of ADs per flow path. Suppose ISPs report at the granularity of ASes; given that 75% of AS pairs are less than 4 ASes apart [24], using 4 as the average number of ADs per flow path seems a reasonable, albeit rough, estimate—note that this number is consistent with the average AS path length observed in current BGP tables [8]. Assuming this number, $5,000$ bytes per flow, and non-split TCP flows as the common case, feedback introduces in each source AD roughly 1.85% bandwidth overhead over the AD's exported traffic. To put this overhead in context, it is worth noting that the IPv6 header would introduce 5% bandwidth overhead, assuming an average packet size of 400 bytes.

[3]Currently, according to data from Route Views [2], more than 99.5% of ASes have fewer than 100 inter-AS connections. ASes with more interconnections would have to be broken to multiple ADs to remain within this marker overhead, without necessarily exposing their internal compartmentalization to feedback receivers.

VIII. Discussion

A. Feedback Tampering

One limitation of our threat model and the presented implementations is the assumption that malicious routers will not selectively tamper with the feedback they observe. In practice, this assumption is justified because an ISP that engages in such feedback tampering is violating legally binding agreements with its peers; moreover, to the best of our knowledge, traffic tampering by malicious on-path routers is not currently known to be a typical Internet problem. However, it could become a problem in the future, unless we provide a way to prevent or expose such malicious behavior.

We are considering two approaches towards dealing with feedback tampering. The simplest one is to enable provider-receiver pairs to detect (but not necessarily localize) feedback tampering, so they can negotiate alternative delivery paths. This can be done by enhancing feedback packets with message authentication codes (MACs). For instance, consider a source AD S, a transit AD X that forwards some of S's traffic, and a malicious entity M on the path from X to S. If M modifies the content of X's feedback packets, S can detect it, as long as each packet carries a MAC. If M drops X's feedback packets, S can detect it, as long as it knows that X is part of the AD-level path and expects to receive feedback from it. In that case, it can issue a follow-up request to X and verify that X did indeed send feedback to S that was dropped along the way.

A more complete, but expensive approach is to force feedback to flow hop by hop through the checkpoints that observed the corresponding traffic. In this way, the AD-level path that delivers each piece of feedback becomes visible, making it possible to investigate feedback-tampering incidents and expose the culprits as with feedback inconsistencies.

B. Flow Sampling

Another limitation is that each checkpoint must observe every single packet in order to produce accurate statistics; in practice, ISPs prefer to use *sampled* NetFlow, which monitors only a configurable percentage of forwarded traffic and, thus, allows them to control the resources spent in monitoring. We are considering adapting our implementation to work with sampled or adaptive [17] NetFlow at the cost of reduced (but bounded) accuracy in the reported statistics.

C. Reflector Attacks

One aspect that we have considered, but left outside this paper for lack of space, is how to prevent reflector attacks. In such attacks, compromised nodes spoof the victim's source address and use it to send a large volume of TCP packets to various destinations, in order to cause the victim to receive a large volume of feedback entries on traffic it did not generate. A key observation that helps us address this problem is that, in order to launch a successful reflector attack, the attacker must generate unusually small (in terms of packets) TCP flows; hence, by packaging feedback on small flows separately, reporting ADs enable feedback receivers to efficiently classify suspicious feedback. Another key observation is that, by studying the feedback received through a reflector attack, the intended victim can trace every single attacking source back to its AD, i.e., a fortuitous side-effect of deploying accountability is that spoofing becomes localizable.

IX. Related Work

Our work was originally inspired by Hash-based Traceback [27]; we share common mechanisms with that architecture, albeit with different goals. In both architectures, traffic leaves a trail on its path. In our case, this trail is per-flow state recorded at the entry and exit points between ADs and is used to send pro-active feedback to source ADs; in Hash-based Traceback, the trail consists of packet digests recorded at each router and is used to send reactive information to destinations.

Accountability has also been studied in the context of Byzantine fault detection in distributed systems [31]: CATS provides accountability for network storage [32], while PeerReview addresses the more general problem of accountability in any distributed system that can be modeled as a collection of deterministic state machines [20]. Both systems provide secure logs of the messages sent and received by each node and identify faulty nodes by processing their logs. Our work is similar in spirit: one can view each AD sequence as a distributed system that keeps "logs" (flow records) of the "messages" (packets) that enter and exit each "node" (AD). Our approach differs mainly in functionality (we measure each node's performance rather than detect Byzantine behavior) and domain specificity: Since our "messages" correspond to packets transmitted over high-speed links, it is still impractical today to produce secure logs of all messages exchanged between nodes. Moreover, because our "nodes" are administrative domains engaged in business with their peers, we do not seek to globally prove a node's misbehavior (in our case, lying) to all other nodes—exposing it to the implicated peer(s) is a sufficient deterrent against misbehavior.

A more theoretical perspective on network accountability is offered in [19]. The authors prove that, to perform accurate fault localization in the presence of malicious entities that can add, drop or modify traffic (including feedback), every feedback provider must share keys with the corresponding feedback receiver and use them to perform cryptographic operations. The same work also presents the Optimistic and Statistical FL (fault localization) protocols, which address a different threat model than AudIt (arbitrary traffic tampering by malicious on-path entities), but are related to the extensions we mention in §VIII: Optimistic FL is similar to the hop-by-hop feedback propagation scheme we mention in VIII-A, although the former performs corruption and loss localization and does so per packet, whereas we are interested in delay and loss localization per aggregate. Statistical FL is related to the sampling approach we mention in VIII-B, in that it considers a subset of the observed packets and estimates the average corruption/loss rate per link; however, it requires sources to trust their destinations and feedback producers to engage in probing sessions with each destination, whereas we are interested in delay/loss localization that is practical to deploy without needing to involve destination hosts or domains.

Another related line of work is Trajectory Sampling, in which routers within an ISP sample packets and record their digests. The key point is that all routers sample the same packets, which allows the ISP to combine the recorded digests and reconstruct its internal paths at a router level [16]. We are considering using this work as a basis to build an alternative AudIt implementation that computes its statistics based on traffic samples. We chose to start with a NetFlow-based implementation instead, only because NetFlow is already widely deployed, whereas Trajectory Sampling requires packet-digesting capabilities on the datapath, which are still unavailable.

In an earlier workshop paper, we describe a preliminary mechanism that informs traffic sources where their packets are getting lost or corrupted [9]. In that work, we take a quite different approach: feedback is sent per packet, not per flow, and it flows hop by hop through the checkpoints that generated the corresponding traffic. The reason for taking that approach was that it allowed every AD on a packet's path (not just the source AD) to receive feedback on the fate of that packet. However, this functionality was provided at the cost of increased bandwidth and memory overhead, processing complexity and the need for custom hardware. In contrast, this paper focused on a practically deployable solution.

Finally, we target similar goals (albeit through different philosophy and mechanisms) with probing tools that seek to localize loss and delay on end-to-end Internet paths. Recent developments include the design of flexible probing processes [28] and scalable algorithms that compute statistics on multiple paths by monitoring only a subset [12]; also, Mao et al. have proposed to complement traditional traceroute by mapping the discovered router addresses to the corresponding ASes, thus producing AS-level paths [25]. From the widely used traceroute program to sophisticated network-tomography techniques [15], probing tools express the traditional (and, admittedly, the only currently applicable) approach to Internet troubleshooting: treat it as a black box and try to guess its internal structure (and faults) by studying its response to different signals. Our approach is the opposite: let the Internet itself (i.e., the ISPs) report on its faults on its own terms, removing the need for probing from multiple vantage points, and avoiding the risk of irritating ISPs into making the black box even more opaque.

X. Conclusions

We proposed AudIt, an accountability interface that enables ISPs to report the loss and delay experienced by transient traffic to the traffic source, while keeping internal ISP structure and policy private. We showed that the proposed interface is resistant to lies in a business-sensible malicious threat model: as long as an ISP follows the reporting interface for some packet aggregate, its peers cannot blame on it their loss and/or delay for the same aggregate without the ISP detecting their lie. We also showed that ISPs can implement AudIt to report on TCP traffic with a modest NetFlow modification and introducing less than 2% of bandwidth overhead on typical Internet traffic.

We believe that an accountability interface would have a positive impact on the Internet. Most importantly, it exposes ISP performance. Good ISPs may want to employ it, to prove to their customers that they are not responsible for packet loss or delay. This may, in turn, drive the remaining ISPs to improve their (now measurable) service. In this sense, accountability could bring better ISP service by increasing competition on performance (which is now only dimly observable), not just on price.

The detailed performance information can also help end systems choose alternate routes to improve their performance. There are many proposals for letting end systems control their routes, but far fewer for how those end systems might gather the information necessary to intelligently choose their routes. By giving them the knowledge of which ADs are currently underperforming, they can narrow their search for better routes. This may even remove the need for Internet QoS mechanisms, since (ignoring access links) there are usually uncongested paths between two network points; to get good quality of service, end systems merely need to find those paths. Our accountability interface, though not a complete solution to this problem, does provide useful information.

The use of layering to hide implementation details from higher layers is a crucial aspect of the Internet architecture; correspondingly, end systems view the Internet as a black box, remaining ignorant of any network structure. But equally crucial is the end-to-end principle of implementing as much functionality as possible at the edges. In particular, Internet applications should adapt to Internet conditions rather than expecting the network to adjust to their requirements. Without more knowledge of the Internet's behavior, the edge's ability to adapt is limited to congestion control and related behavior.

Our accountability interface is designed to provide structural information out of band. It preserves layering and leaves IP semantics unchanged; it is an external vehicle for informing the host of network conditions. Many have called for an Internet knowledge [14] or information plane [29] that would expose network information to end systems. We view this work as a first concrete and viable step in this direction.

XI. Acknowledgments

We would like to thank Steven Dropsho, Aravind Menon, Simon Schubert, Ming-Yee Iu, Matt Grossglauser, and the anonymous reviewers for their insightful comments that helped improve this paper. This research was supported by a Marie Curie International Reintegration Grant within the 6th European Community Framework Programme.

References

[1] Cisco NetFlow. http://www.cisco.com/go/netflow.
[2] Route views archive project. http://archive.routeviews.org.
[3] The CAIDA Web Site. http://www.caida.org.
[4] USNO GPS Time Transfer. http://tycho.usno.navy.mil/gpstt.html.
[5] Cisco 12410 and Juniper M160 Comparison Summary Report. http://newsroom.cisco.com/dlls/Cisco12400JuniperM160PerfVal.pdf, June 2001.
[6] Packet Reordering in Juniper M160. http://www.lightreading.com/document.asp?doc_id=4009&page_number=8, March 2001.
[7] Juniper Goes Terabit with the T640. http://www.lightreading.com/document.asp?doc_id=14335, April 2002.
[8] BGP Table Data. http://bgp.potaroo.net/as6447, August 2007.
[9] K. Argyraki, P. Maniatis, D. R. Cheriton, and S. Shenker. Providing Packet Obituaries. In *Proceedings of the ACM Workshop on Hot Topics in Networking (HotNets)*, November 2004.
[10] D. J. Bernstein. Syn cookies. http://cr.yp.to/syncookies.html.
[11] J. Burbank, W. Kasch, J. Martin, and D. Mills. Network Time Protocol Version 4 Protocol and Algorithms Specification. http://tools.ietf.org/html/draft-ietf-ntp-ntpv4-proto-06, May 2007.
[12] Y. Chen, D. Bindel, H. Song, and R. H. Katz. An Algebraic Approach to Practical and Scalable Overlay Network Monitoring. In *Proceedings of the ACM SIGCOMM Conference*, September 2004.
[13] D. R. Cheriton and M. Gritter. TRIAD: A Scalable Deployable NAT-based Internet Architecture. Technical report, Stanford University, January 2000. Also available at http://www.dsg.stanford.edu/triad/triad.ps.gz.
[14] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A Knowledge Plane for the Internet. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.
[15] M. Coates, A. O. Hero, R. Nowak, and B. Yu. Internet Tomography. *IEEE Signal Processing Magazine*, 19(3):47–65, May 2002.
[16] N. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. *IEEE/ACM Transactions on Networking*, 9(3):280–292, June 2001.
[17] C. Estan, K. Keys, D. Moore, and G. Varghese. Building a Better NetFlow. In *Proceedings of the ACM SIGCOMM Conference*, September 2004.
[18] A. Feldmann, A. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. *IEEE/ACM Transactions on Networking*, 9(3):265–280, June 2001.
[19] S. Goldberg, D. Xiao, B. Barak, and J. Rexford. A Cryptographic Study of Secure Internet Measurement. Technical Report TR-783-07, Princeton University Department of Computer Science, May 2007.
[20] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, October 2007.
[21] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *IEEE/ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.
[22] A. Kuzmanovic and E. W. Knightly. Low-Rate TCP-Targeted Denial of Service Attacks. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.
[23] P. Laskowski and J. Chuang. Network Monitors and Contracting Systems. In *Proceedings of the ACM SIGCOMM Conference*, September 2006.
[24] D. Magoni and J. J. Pansiot. Analysis of the Autonomous System Network Topology. *ACM SIGCOMM Computer Communication Review*, 31(3):26–37, July 2001.
[25] Z. M. Mao, J. Rexford, J. Wang, and R. H. Katz. Towards an Accurate AS-Level Traceroute Tool. In *Proceedings of the ACM SIGCOMM Conference*, August 2003.
[26] B. Raghavan and A. C. Snoeren. A System for Authenticated Policy-Compliant Routing. In *Proceedings of the ACM SIGCOMM Conference*, September 2004.
[27] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP Traceback. In *Proceedings of the ACM SIGCOMM Conference*, August 2001.
[28] J. Sommers, P. Barford, N. Duffield, and A. Ron. Improving Accuracy in End-to-end Packet Loss Measurement. In *Proceedings of the ACM SIGCOMM Conference*, August 2005.
[29] M. Wawrzoniak, L. Peterson, and T. Roscoe. Sophia: An Information Plane for Networked Systems. In *Proceedings of the ACM Workshop on Hot Topics in Networking (HotNets)*, November 2003.
[30] X. Yang. NIRA: A New Internet Routing Architecture. In *Proceedings of the ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA)*, August 2003.
[31] A. R. Yumeferendi and J. S. Chase. The Role of Accountability in Dependable Distributed Systems. In *Proceedings of the IEEE Worshop on Hot Topics in Dependable Systems (HotDep)*, June 2005.
[32] A. R. Yumeferendi and J. S. Chase. Strong Accountability for Network Storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST)*, February 2007.
[33] X. Zhang, A. Perrig, and H. Zhang. Availability-Oriented Path Selection in Multi-Path Routing. Technical Report CMU-CyLab-07-012, Carnegie Mellon University, August 2007.