Net-Replay: A New Network Primitive

Ashok Anand, Aditya Akella
University of Wisconsin-Madison
{ashok,akella}@cs.wisc.edu

ABSTRACT

In this paper, we describe *Net-Replay*, a new network primitive to help application end points conduct in-band characterization of the glitches they encountered. In *Net-Replay*, each network infrastructure element remembers a small amount of information for every packet observed at the element over a certain time interval. Furthermore, network elements expose a simple "packet marking" interface, using which they can indicate to end-points whether or not they had seen a particular packet in the past. When application end-points observe glitches, they *replay* (i.e. retransmit) the packets which observed the glitch and leverage feedback from network elements to determine the type and location of the glitch encountered by the packets. We discuss how end-host network stacks should be modified to leverage *Net-Replay* in this fashion. We also consider how network infrastructure can support *Net-Replay* in a low-overhead fashion.

We argue that *Net-Replay* can enable applications to detect a variety of glitches and react to them in an accurate and informed manner, while ensuring that the infrastructure stays simple and fast. We believe that proactive support from the network in the form of *Net-Replay*-like functionality is crucial to ensure robust performance of future Internet applications, many of which are likely to be highly demanding and far less tolerant of network glitches than traditional applications.

1. BACKGROUND AND MOTIVATION

The network infrastructure today does not support any way to provide information regarding on-going transmissions to end applications. This information is maintained in some adhoc manner (e.g. netflow etc) for some network devices, but is not available to network-end points via any standardized interface. This helps keep the network simple and efficient.

However, this design choice has had a significant impact on applications, in particular, on how application flows detect and respond to *network-induced glitches*. End-to-end flows could be affected by a variety of glitches, such as packet drops, re-orderings or delays. Such glitches could have a significant impact on user experience, especially when performance-sensitive applications such as streaming video, VoIP etc. are employed. Since the network does not tell anything regarding end-to-end transfers, application end-points (in particular, the transport protocols) have to resort to *complicated logic* to infer *what* had impacted their transfers so that they can react appropriately to network glitches. In many cases, it is difficult to make an accurate inference — for example, congestion losses cannot be distinguished easily from losses due to packet errors or packets dropped by on-path filters — causing application flows to react erroneously and affecting user experience. Also, owing to the current design, application flows have no information regarding *where* in the network their packets had encountered a glitch. If available, the application logic at end-points could use the knowledge to *overcome* the glitches by routing around them (using route-control mechanisms [1, 2], or recently-proposed protocols enhancements [12, 14, 9]), and improve user experience.

We believe that emerging demanding applications will hasten the need for a more *proactive* involvement from the network. The recent years have seen tremendous growth in applications like VoIP, gaming, e-commerce applications, and streaming over the WAN, which are far less tolerant of network glitches and demand much more robust and resilient network behavior than traditional elastic flows. Given these trends, relying solely on network end-points to salvage robust end-to-end experience in the face of network-induced glitches is likely to make application and transport protocol designs more and more complex. *Yet*, there is no guarantee that the stringent needs of modern applications will be met.

In this paper, we propose a candidate approach for adding proactive support into the network. In our approach, the network supports a new primitive called *Net-Replay* to help *ongoing application flows* determine the type of glitches their data packets had encountered as well as the location of the glitch. End-points could use the improved knowledge of network glitches at the transport and/or application layers to respond to, or to overcome, glitches in an informed fashion. Our insight is that *Net-Replay* can be realized by implementing a couple of extremely simple mechanisms in network elements and installing a small amount of logic at network end-points. Thus, the simplicity and speed of the network infrastructure is not sacrificed in any way.

In *Net-Replay*, network infrastructure elements perform two simple functions:(1) Network elements remember set of packets that they have *successfully forwarded* over a past time interval, irrespective of the application, source, destination or protocol fields in the packets. (2) Network elements expose a simple "packet marking" interface; Before forwarding packets, network elements annotate them to indicate if they had *never seen* the packet in the near past. When flow senders detect that some glitch had occurred (based on TCP ACK packets or other special reception reports), the senders re-transmit, or *replay*, a small number of additional packets which are exact duplicates of the packets that had experienced the glitch. When the replayed packets arrive at the app-receiver, the annotations in them

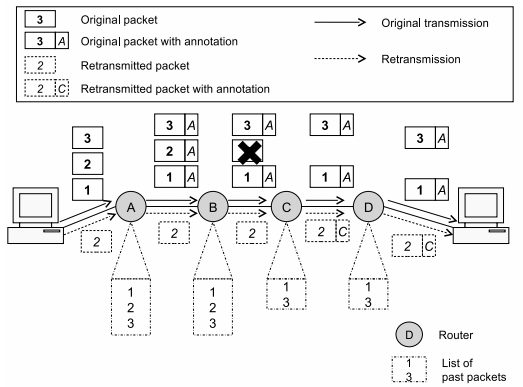

Figure 1: Using *Net-Replay* to characterize packet loss: (1) The sender application transmits three packets. (2) On-path routers remember the packets locally. Since router A saw the packets for the first time, it annotates all of them with its identifier. Downstream routers don't annotate any further. (3) Router B drops packet 2. (4) Receiver detects packet 2 was dropped and lets sender know (not shown). (5) Sender replays packet 2 (shown using dashed arrows). (6) Router C checks against its local store and observes that it never saw packet 2 earlier. C annotates the retransmission. (7) The receiver conveys the information to the sender. The senders knows that the router upstream from C caused the drop.

can be used to derive the type and location of the glitches encountered by the *original packets*. The whole process takes a handful of extra RTTs. Figure 1 shows an example of how application flows can use *Net-Replay* to characterize packet loss. Thus, *Net-Replay* allows an application flow to conduct *in-band investigation* of the glitches experienced by a *specific earlier packet* in the flow.

There exist a few probing-based approaches today to characterize network-induced glitches (see, for example, tulip [7]). However, they suffer from key drawbacks which prevent them from supporting *Net-Replay*-like functionality effectively. First, in current approaches, the probing is done *out-of-band* and *after* a glitch has been observed. Second, the network may treat probe packets differently than data packets. Third, most probing techniques are ineffective at detecting transient glitches. Many aspects of our design of *Net-Replay* are informed by these drawbacks.

Net-Replay is a network-supported but application- and transport protocol-driven framework for diagnosing network glitches. In-fact network elements need not remember packets of all applications, and can remember packets of only those applications who want to leverage *Net-Replay*. Applications can indicate so by using a flag bit in the packet header. However, for simplicity, we consider network elements to remember packets from all application flows in rest of our discussions. In Section 3, we discuss in detail the network support required for *Net-Replay*.

We argue that *Net-Replay* will enable application flows to accurately identify and respond to a variety of network glitches, including those which are very short-lived. We also argue that *Net-Replay* can be incorporated into end-host network stacks in a way that helps make both application and transport protocol designs flexible and simple. Applications are given flexibility to decide how they want to *overcome* glitches, while mechanisms are built into transport layer to *detect and characterize* glitches. This retains the simplicity of transport layer and gives flexibility for application designers. In particular, a variety of flexible end-point strategies can be developed to *overcome* glitches. End-hosts could leverage recent end-host controlled routing protocols enhancements [13, 12, 14, 9] to overcome the glitches. Others may decide to track historical performance of ISPs and use it to avoid some ISPs altogether. We discuss this in more detail in Section 4.

We also present an approach for incrementally deploying *Net-Replay* into a network with legacy infrastructure. We argue that *Net-Replay* could be deployed within high-speed two ports switches, which we refer to as "bumps in the wire", around the network devices. We provide further details in Section 5.

We believe that the key ideas from our proposal can also be applied to develop debugging tools for specialized settings, such as large complex distributed systems, data centers and enterprise networks. Augmenting infrastructure with memory and exporting an *Net-Replay*-like interface can help shed light on some complex failure modes in these settings.

2. RELATED WORK

Our ideas behind *Net-Replay* are motivated by a variety of past studies. We review these below.

Argyraki et al. propose a framework which relies on each ISP to provide regular, out-of-band feedback to senders regarding the performance of an application's packets as the packets traverse the ISP network [3]. Each ISP is required to track ISP-wide performance (e.g. packet loss, delay etc.) experienced by application flows using expensive state on network devices. To reduce the amount of overhead due to ISP reports, Argyraki et al. propose to offer aggregate-level feedback (generated either at long time scales or for groups of flows). Also, the proposal requires the support of a special framework to help end-hosts ensure the authenticity of reports generated by on-path ISPs. In contrast *Net-Replay* is relatively light-weight, requires no expensive state and does not rely on out-of-band communication originating from network devices. Yet, *Net-Replay* can allow any individual application flow to characterize the network glitches it had experienced and respond appropriately.

The idea of storing packets in routers to figure out whether they were observed earlier has been used in Snoeren et al.'s work on single packet-based IP traceback [11]. Similar to our study, Snoeren et al.'s work also uses Bloom filters to minimize storage overhead on routers. Our study differs in several other key aspects, however: in particular, we go well beyond Snoeren et al.'s framework and extend it to have all routers expose an in-band packet marking interface universally to all applications. We describe how regular end-hosts and applications can leverage the interface to improve their end-to-end experience. Packet marking has been explored in several other contexts as well, including Savage's original traceback work [10] where *probabilistic* marking was considered, and the ECN proposal where the network indicates incipient congestion through marks (but not where the congestion was occuring). Our proposal can be viewed as generalizing both of these marking techniques.

The recent X-trace proposal [6] also bears some similarity to our work. X-trace logs the information associated with various network operations, focusing in particular on tracking causal dependencies between different protocols, applications, and middleboxes which may come into play during an end-to-end communication. The goal is to use the logs to perform retroactive diagnosis of failures. In contrast, *Net-Replay* enables in-band investigation of glitches faced by ongoing communications. The second key difference is that *Net-Replay* is end-to-end application in nature, in that the network does not generate any feedback unless applications elicit it explicitly. In contrast, X-trace is more provider-centric in that the expectation is that the device logs are analyzed ex post facto for possible causes of failures.

Orchid [8] has similar goals of in-band network fault diagnosis.

However, Orchid requires expensive per-flow state maintainance at each router. Many of the details are not there e.g. interactions with end-hosts is not clear. In contrast, *Net-Replay* is relatively simpler. We have explored the end-host interactions in greater detail, and many other issues.

3. NETWORK SUPPORT FOR *Net-Replay*

In this section, we describe the functionality necessary within network elements to support *Net-Replay*. We focus our description on network routers, although our arguments apply at a high level to other network elements.

Basic Functionality. To support *Net-Replay*, each network router needs to remember a list of all packets that it has successfully forwarded in a past time-window. A router stores a hash for each such packet in *HashStore*, where the hash is computed over the entire content and header, minus the mutable fields of the header like the TTL. As a packet traverses a network route, each router checks if the packet is present in its local HashStore. If packet is *not found*, implying that the router did not observe the packet over a past time interval, then the router appends its *identifier* (e.g. address of the incoming interface) to the packet. If multiple routers append their identifiers then packet sizes can grow arbitrarily. So we insert the identifier of just the first such router into a single fixed-size field in the packet header; we call this the "PacketNotFoundAt" field. If downstream routers notice that the field is non-empty then they simply forward the packet along, without checking for the packet's presence in their local hash table. This helps expose the first location on the path where a glitch was observed.

Practical HashStore Implementation. To support *Net-Replay* effectively, routers need to store packet hashes and retrieve them fast enough. On a 2.5 Gbps router, a new packet can arrive every 128 ns (minimum 40B packet). This is enough time for storing and retrieving hashes, from DRAM of latency 50 ns. So HashStore can be stored in DRAM at these speeds using simple hashtables. 1 GB DRAM would be enough to store around 25s worth packets using 16B hashes for 600B average packet size.

However for higher speeds like 40 Gbps, HashStore must be stored in faster memory like SRAM. But, with current limit of 16 MB SRAM, just few milliseconds worth packets would be stored. So an immediate option is to consider space-efficient data structures like *Bloom filters*. Similar to Snoeren et al.'s proposal [11], we can use a Bloom filter with three hash functions ($k = 3$) and a capacity factor (m/n) of five, i.e. bloom filter of size $m = 5n$ bits for storing $\leq n$ unique packets. Such filter has a false positive rate of $P = 0.092$ when n keys have been inserted. In case of false positives, a router would think that it has seen a packet earlier when it has not. This could mean that precise point of loss can be misidentified. However, the imprecision will be within one hop of actual loss-point with probability $(1 - P^2) = 0.9915$, which is reasonably high. End applications can take into account of this small imprecision while taking decisions for working around glitches.

Stale entries needs to be cleared from bloom filters. So, we propose using *two* bloom filters in tandem, where one of the bloom filters (called Secondary) lags in time behind the other (called Primary). In particular, when a certain number of bloom filter entries are set in the Primary (corresponding to insertions of $n/2$ packets), we start populating the secondary (which is empty up until then). When the Primary is filled up with n entries, we copy the contents of the secondary into the primary, and flush the secondary.

With two bloom filters, a 16MB SRAM can store around 3s worth packet hashes for a 40Gbps router, assuming an average packet size of 600B. This is more than 10 RTTs for $RTT < 250ms$. As we will argue later, this gives enough time to characterize network-induced glitches using transport layer mechanisms. Routers can also periodically dump their SRAMs to slower memory to provide a log for future queries, to aid operators in traffic engineering tasks.

More Information Per Packet. Storing additional information per packet can help applications characterize and respond to more sophisticated network-induced glitches. Along with the packet hash, each router can store meta-data regarding the queuing delay incurred by the packet, or the spare capacity on the outbound link, just before the packet was forwarded along. Bloom filters do not provide any support for storing meta-data, but we can have another bloom filter for specific metadata – for example, we can have another bloom filter for storing packets which had experienced a high queuing delay (greater than some *threshold* value). Then in similar manner, location of delay can be determined.

4. END-HOSTS USING *Net-Replay*

In this section, we describe how the *Net-Replay* primitive can be leveraged by end-hosts and applications to characterize a variety of network-induced anomalies, including packet drops, reordering and excessive queuing delay. Later in this section, we describe where in the end-host's networking stack the schemes described below should be implemented. In particular, we argue that the transport protocol should implement the techniques for characterizing the anomalies.

Throughout the description below, we assume that the sending host detects a glitch based on feedback from the receiver (e.g. via duplicate acknowledgments, timeouts, receptions reports in RTSP etc.) and also knows which packets seem to have experienced the glitch. We focus on describing how the sender goes about characterizing the glitch.

Recall, as we mentioned earlier, that application end-points replay the packet(s) that they think experienced a glitch to determine what happened to the packet(s).

4.1 Characterizing Glitches

Packet loss. Upon detecting a lost packet, the sender replays or retransmits the missing packet exactly, in full. Based on the PacketNotFoundAt field in the replayed packet, the receiver can know which router dropped the packet (namely, the router upstream from the one in the PacketNotFoundAt field). The receiver can relay this information back to the sender.

Note that it is possible that a route change took place by the time the sender replayed the missing packet. This could provide false information regarding which router dropped the packet. To overcome this possibility, the sender can replay one *non-missing* packet along with the missing packet. If a route change did take place, the replayed non-missing packet will also be seen by some router for the first time and will carry a mark. If a route change did not take place then the replayed non-missing packet will not carry any marks while the replayed missing packet will carry the location of the router that had dropped it.

Packet Reordering. Upon knowing which packets got reordered (based on receiver feedback), the sender replays the re-ordered packets in full. Suppose that the reordering occurred due to a route change. If the route has changed, similar to packet drop scenario above, the first router where the route change took place can be inferred from the marks in the replayed packets. Now suppose that the reordering occurs due to other reasons, a common one being some intermediate route preferentially forwarding small packets before large packets. In this case, the replayed packets will also arrive out

of order at the receiver (just as the original packets had done), but they will not carry any marks.

Delay. Suppose that the receiver informs the sender that some packet arrived in a highly delayed fashion, e.g. a voice or video sample may have arrived much after its play-back deadline, or a TCP receiver may have noticed that inter-packet arrival times are high. The sender then replays the delayed packets. Beyond this, the approach is the same as described earlier, with the receiver relaying to the sender the network location where the highest delay was encountered.

We note here that sender can also use *Net-Replay* to determine, in some scenarios, if a packet drop was due to congestion or due to non-congestion-related reasons. To do this, the sender can examine if high delays were observed by packets prior to the dropped packet, which may indicate an impending buffer overflow. This is applicable if the sender is transmitting packets at a fast rate. Otherwise, the packets prior to the dropped packet may have been transmitted at a time when the queue was just beginning to build, and hence they may have observed no serious delay at all.

4.2 Division of Functionality

We discuss how end-point protocol stacks should be modified to leverage *Net-Replay*. In particular, we focus on the split-up of functionality between transport and higher layers.

Since *Net-Replay* deals with glitches observed by data packets and the network elements also store information regarding individual packets, we believe that it is best to incorporate all the mechanisms required to *detect and characterize* glitches using *Net-Replay* into network *transport protocols*.

The logic for whether or not to *overcome* the glitches, and how to overcome them, can also be implemented at the transport layer. But we argue that this is a policy decision and hence it must be implemented within the *higher layers*. Here, we use the term "higher layers" liberally to include both applications as well as the end-networks where the applications run. For instance, some applications and end-networks may decide that the glitches don't deserve any action unless they are severe and persistent. Other applications or end-networks may track the incidence of glitches over time, and decide to use routes which avoid the most troublesome network locations. Dividing functionality in this manner gives application designers the flexibility to incorporate a variety of strategies to overcome glitches, without having to deal with developing algorithms for detecting the glitches themselves. At the same time, it also frees the transport protocols from having to incorporate a common set of strategies for overcoming glitches that match a variety of application needs.

We note that in order to effectively support higher-layer strategies, the *interface between applications and transport protocols* must be enriched so that the inferences derived by transport protocols could be exposed to higher layers. Below, we use TCP as an example and discuss how it must be modified to detect and characterize glitches, and how to enrich the interface between TCP and the higher layers.

Modifying TCP. We briefly discuss how to modify TCP senders and receivers. We focus on characterizing packet losses as an example. Recall that several modern TCP versions implement the fast retransmit algorithm where the TCP sender waits for three duplicate acknowledgments to detect a packet loss. Subsequently, the sender reduces the congestion window in half as part of TCP's congestion control, and retransmits the packet. Our insight is that the retransmission following a packet loss can be treated as a packet replayed for diagnostic purposes.

More specifically, the fast retransmit algorithm can be modified slightly as follows to work with *Net-Replay*. As with current TCPs, the TCP sender retransmits the lost packet and cuts the congestion window in half. The retransmitted packet arrives at the receiver with annotations inserted by on-path network elements. Upon receiving the retransmitted packet, the receiver, as usual, sends an ACK for the highest in-sequence packet. In addition, if the receiver is *Net-Replay*-aware, it creates another spurious ACK (TCP flags can be used to indicate that this is a spurious ACK), into which it incorporates the annotations received in the retransmitted packet along with a hash of the original lost packet; thus, the spurious ACK is slightly larger than a traditional TCP ACK packet. If the receiver is not *Net-Replay*-aware, then it simply ignores the annotations in the retransmitted packet and the sender will not be able to characterize the loss.The annotations reflected by the receiver in the spurious ACK will indicate the location and cause of loss. The sender TCP exports this information to the higher layer.

We note that since *Net-Replay* provides richer information regarding the *nature* of the loss, TCP's congestion control algorithm can use it to determine *how best to react to a packet drop*. For example, as described above, if annotations indicate that packet loss was not due to congestion, TCP sender can *re-open* congestion window back to size prior to loss.

Interface between TCP and Applications. We discuss briefly how the interface between TCP and applications can be modified so that TCP can push "upward" all the diagnostic information to applications that may want to use it. One way to achieve this is to allocate a small amount of memory on the sending host as "scratch space", into which TCP can write its inferences and the relevant application can read them. For instance, TCP can create a log with a list of entries of the form <type of glitch, time of glitch, location of glitch>. The scratch space can be allocated alongside the memory allocated for a transmission control block (TCB) when intiating a TCP connection.

Response from Higher Layers. We discuss how higher layers can overcome network glitches (if they so desire) based on inferences provided by transport layers. In particular, we discuss how the feedback can be used along-side recent proposals for intelligent route control.

Given multiple options and ability to choose a path, the end point can select routes which avoid the location of the glitches. Multihoming provides multiple paths but end-point can only select paths from its ISPs and does not have any control over the path that ISP selects for a destination. MIRO [12] provides more explicit control over the AS path; the feedback from *Net-Replay* can be coupled with MIRO to avoid troublesome ISPs. Recent "Path Splicing" [9] and "route deflections" [14] proposals go well beyond multihoming and MIRO, and provide much greater flexibility for end-points to select paths allowing them to switch paths at any intermediate hop along the path. All of these proposals can be leveraged alongside *Net-Replay* to overcome glitches.

5. DISCUSSION

In this section, we discuss some practical deployment issues, incentives of ISPs, challenges due to cheating and other applications of *Net-Replay*.

5.1 Deployment

Coarser Information. In the form described earlier, the information provided by *Net-Replay* was at the granularity of an individual device. Some network infrastructure providers and ISPs may not wish to reveal internal information to application end-points at such fine-granularity for security and competitive reasons. *Net-Replay*

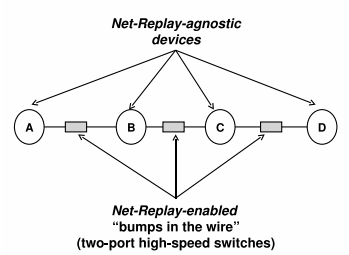

Figure 2: An illustration of "bumps in the wire".

can be easily extended to provide information at the granularity of a single domain.

In-fact, it is also possible for a mixture of modes of operation, where some domains on a network-path provide feedback at a very fine-granularity (individual device) while others provide information at the level of a domain.

Also, note that even if a service provider only exposes coarse-grained information to network end-points, it can *internally* employ fine-grained router-level markings on end-point-replayed packets to debug delay and loss issues inside its own network and conduct traffic engineering effectively. The fine-grained router marks can be converted to coarse grained marks as packets exit the network.

Partial Deployment. *Net-Replay* could also provide benefits when deployed on a subset of network devices. In particular, ISPs could enable this functionality on just their borders routers and this is still sufficient to provide feedback to applications at the granularity of a domain. *Net-Replay* can also be enabled on error-prone devices and devices with complex failure modes (and their neighbors).

Avoiding Device Modifications. *Net-Replay* requires modifications to network devices and this could be prohibitive from the view-point of some network device vendors. We claim that devices modifications can be avoided altogether and *Net-Replay* can be applied alongside off-the-shelf network infrastructure. To see how, note that instead of modifying a device, we can deploy *Net-Replay*-enabled "bumps in the wire" in the vicinity of the device. The "bumps" are high-speed two-port hardware switches which perform the simple functions of remembering which packets were successfully forwarded, and marking packets which were seen for the first time with the identity of the upstream device. An example deployment of the bumps is shown in Figure 2.

5.2 Incentives and Cheating

We believe that service providers would have a natural incentive to deploy *Net-Replay*, especially in the coarse-grained form described above. This is because, *Net-Replay* feedback can be leveraged by service providers to provide *accounting* for end-to-end transfers, which subscribers may view as an attractive value-added service.

Net-Replay could potentially suffer from issues related to cheating: a service provider network could insert wrong annotations into packets to make it seem like an upstream network introduced glitches (e.g. losses), whereas in reality its own infrastructure was responsible for the glitches. ISP could do so for a significant fraction of traffic to ensure that it is not considered accountable for any of its glitches. In that case there is a high chance that ISP will be caught, and this could deter ISP from cheating. Another possibility is that an ISP can delete annotations on the return path to the sender that implicate it as having dropped packets. ISP can inspect the "spurious" TCP ACKs, search for its own routers' IP addresses within them and drop them. However, this can be easily avoided by sending *encrypted* spurious TCP ACKs, and ISP would not know if it is being considered responsible for dropping packets.

We are currently investigating mechanisms for detecting and thwarting several such cheating instances.

5.3 Other applications

Since *Net-Replay* enhances the feedback provided by the network, it helps simplify the design of many current troubleshooting tools and applications, and also enables new ones. Consider, for example, network tomography for inferring link loss rates in a given network topology using a small collection of end-to-end measurements. Senders either send multicast probes or back-to-back unicast probes and receivers measure the loss rates. Since the location of the loss cannot be determined, existing proposals [4, 5] employ a variety of sophisticated statistical techniques to assign the most probable link-loss rates for various internal network links. *Net-Replay* simplifies network tomography to a great extent. Whenever there is a loss of probe packet, the sender replays the probe packet to find the exact location of the loss. There is no need to infer the location where the loss is most likely to be occurred. Thus, the link-loss rate can be easily and more precisely determined.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we described *Net-Replay*, a new network primitive to help application flows conduct in-line characterization of the glitches they encountered. We discussed the support required with infrastructure elements as well as how end-point stacks must be modified to leverage *Net-Replay*. We argued that *Net-Replay* can enable applications to detect a variety of glitches and react to them in an informed fashion. We strongly believe that *Net-Replay*-like support from the network is crucial to ensure robust performance of future Internet applications.

We have left several issues unaddressed or partially addressed in this work. One such issue is how unreliable network protocols such as UDP can be modified to leverage *Net-Replay*. Even in the context of TCP, it is important to understand how exactly to modify TCP end-points to react to some complex problems, such as pathological reordering and a mixture of loss, delay and reordering (we only considered the example of packet drops). A final issue is to explore the countermeasures for deterring/detecting cheating by ISPs.

7. REFERENCES

[1] A. Akella, S. Seshan, and A. Shaikh. Multihoming performance benefits: An experimental evaluation of practical enterprise strategies. In *USENIX 04*.
[2] D. G. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. N. Rao. Improving web availability for clients with monet. In *NSDI*, 2005.
[3] K. J. Argyraki, P. Maniatis, O. Irzak, S. Ashish, and S. Shenker. Loss and delay accountability for the internet. In *ICNP*, 2007.
[4] R. Cáceres, N. G. Duffield, J. Horowitz, D. F. Towsley, and T. Bu. Multicast-based inference of network-internal characteristics: Accuracy of packet loss estimation. In *INFOCOM*, 1999.
[5] N. G. Duffield, F. L. Presti, V. Paxson, and D. F. Towsley. Inferring link loss using striped unicast probes. In *INFOCOM*, 2001.
[6] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *NSDI*, 2007.
[7] R. Mahajan, N. T. Spring, D. Wetherall, and T. E. Anderson. User-level internet path diagnosis. In *SOSP*, 2003.
[8] M. Motiwala, A. Bavier, and N. Feamster. Network troubleshooting: An in-band approach. In *NSDI*, 2007.
[9] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *SIGCOMM*, 2008.
[10] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical network support for ip traceback. In *SIGCOMM 2000*.
[11] 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 *SIGCOMM*, 2001.
[12] W. Xu and J. Rexford. Miro: multi-path interdomain routing. In *SIGCOMM*, 2006.
[13] X. Yang, D. Clark, and A. W. Berger. Nira: a new inter-domain routing architecture. *IEEE/ACM Trans. Netw.*, 15(4), 2007.
[14] X. Yang and D. Wetherall. Source selectable path diversity via routing deflections. In *SIGCOMM*, 2006.