PeriScope
An Active Measurement API

Khaled Harfoush harfoush@cs.bu.edu
Azer Bestavros best@cs.bu.edu
John Byers byers@cs.bu.edu

Computer Science Department
Boston University
Boston, MA 02215

Abstract— **Growing interest in inference and prediction of network characteristics is justified by its importance for a variety of network-aware applications. One widely adopted strategy to characterize network conditions relies on *active, end-to-end* probing of the network. Active end-to-end probing techniques differ in (1) the structural composition of the probes they use (e.g., number and size of packets, the destination of various packets, the protocols used, etc.), (2) the entity making the measurements (e.g. sender vs. receiver), and (3) the techniques used to combine measurements in order to infer specific metrics of interest. In this paper, we present Periscope: a Linux API that enables the definition of new probing structures and inference techniques from user space through a flexible interface.** PeriScope **requires *no* support from clients beyond the ability to respond to** ICMP ECHO REQUESTs **and is designed to minimize user/kernel crossings and to ensure various constraints (e.g., back-to-back packet transmissions, fine-grained timing measurements) We show how to use Periscope for two different probing purposes, namely the measurement of shared packet losses between pairs of endpoints and for the measurement of subpath bandwidth. Results from Internet experiments for both of these goals are also presented.**

Keywords— **End-to-end measurement; loss rate; bottleneck bandwidth; Bayesian probing; Cartouche probing; network-aware applications.**

I. Introduction

Measurement of various network characteristics (e.g. loss, delay, bandwidth) is crucial for many Internet applications and protocols, especially those involving the transfer of large files and those involving the delivery of content with real-time QoS constraints, such as streaming media. Examples of the importance of bandwidth estimation include request routing protocols in Content Distribution Networks (CDNs) [1] or in Peer-to-Peer (P2P) networks [33], network-aware cache/replica placement and maintenance policies [20], [31], flow scheduling and admission control policies at massively-accessed content servers [9], end-system multicast and overlay network reconfiguration protocols [8], [19], [2], among many others.

Current network characteristics inference strategies can be classified into two broad categories: The first strategy is to mine the data collected by network internal resources, such as BGP routing tables, to generate performance reports [16], [25], [7], [17]. This approach is best applied over long-time scales to produce aggregated analyses such as Internet weather reports, but does not lend itself well to providing the timely requirements of network-aware applications. The second strategy is statistical inference of network internal characteristics based on measurements obtained from *probing* network resources (routers and/or endhosts) [5], [6], [34], [21], [32], [29], [26]. Using this strategy, information is gathered at the appropriate time scale to address timely network-aware application requirements. A probing approach can be further classified as *active*, i.e. it introduces additional probe traffic into the network, and *passive*, i.e. it makes inferences only from existing network traffic. The benefit of the former is flexibility: one can make measurements at those locations and times which are most valuable; while the benefit of the latter approach is that no additional bandwidth and network resources are consumed just for the purpose of data collection.

Paper Contributions: In this paper, we adopt the active probing strategy. Specifically, we present a Linux API (called PeriScope[1]) that implements the

This work was partially supported by NSF research grants ANI-9986397, ANI-0095988 and CAREER ANI-0093296.

[1]**P**robing **E**ngine for the **R**ecovery of **I**nternet **S**ubgraphs

functionality necessary to define, activate and collect measurements for arbitrary probing structures. To the best of our knowledge, there is no publicly available framework that has PERISCOPE flexibility in defining arbitrary probing structures. For example, by defining a probing structure consisting of a pair of probe packets of the the same size destined to the same endpoint, PERISCOPE can infer the bottleneck bandwidth along a path [7], [27], [29], [28]. By directing the probe packets to different endpoints, PERISCOPE can infer the loss rate along the shared path, induced through IP routing, connecting the probing server to the endpoints [14]. By defining a probing structure consisting of a pair of different size probe packets, PERISCOPE can infer the capacity bandwidth of every physical link along the path [22], [23]. Also, by defining a probing structure consisting of a sequence of different size and destination packet-pairs, PERISCOPE can efficiently infer the bottleneck bandwidth along arbitrary path segments and along the shared paths between different endpoints [15]. PERISCOPE can also easily be extended to provide additional functionality.

Using the PERISCOPE API, we have developed a tool to infer and label the loss, delay and bandwidth topologies connecting a server to a set of clients. These topologies are instantiations of the *Metric-Induced Network Topologies* (MINT) framework, which aims at providing *compact* and *efficient* respresentations of network resources. A metric-induced topology is a labelled logical topology, parametrized with a sensitivity parameter c which is the minimum value of a label which can be applied to an internal link in the topology. For more details about the MINT framework refer to [4].

Ideally, PERISCOPE functionality should be installed at both the server and the client sides. Client-side PERISCOPE installation is only intended to intercept sever probe packets and record their *relevant* characteristics. However, acquiring control over all potential clients is not always possible. It is thus desirable to communicate probe packets characteristics from the clients back to the server side which could be done through replies to ICMP ECHO REQUEST probe packets. The problem with this approach is that the expected characteristics of the probe packets that are delivered to the clients may be altered either at the client itself or over the back channel, on the way back to the server. This may result in inaccurate estimates which is the price one has to pay if we are to deploy PERISCOPE only at the server side.

In this paper we describe the details of the server-side PERISCOPE architecture which, as discussed above, is a superset of the client-side architecture.

Paper Overview: The rest of this paper is organized as follows: In sections II and III, we review existing probing literature and provide the probing terminology that we use throughout the paper. In section IV, we discuss the rationale behind PERISCOPE design and in sections V and VI, we describe PERISCOPE architecture and its user level API. In section VII, we describe some of our experiences with Internet validation of PERISCOPE, drawing on examples taken from a large set of experiments that are aimed at inferring from end-to-end measurements both the loss rate and the bottleneck bandwidth along the shared path connecting a server to a set of clients.

II. RELATED WORK

As noted in the introduction, PERISCOPE relies on actively probing network resources in order to estimate network properties. In general, there are two approaches to implement probing; we can do it (1) in user space [24], [11], [23] or (2) in the kernel [12].

Kernel support is desirable because probing techniques require quality of service guarantees from the probing host which cannot be provided from user space. For example, the uniform packet-pair probing technique requires that packets of each pair be injected back-to-back in the network, and we discuss why this cannot be done easily from user-space.

Acquiring probing functionalities in the kernel can be achieved using two possible approaches: (a) hand-code kernel functionality [12], or (b) deploy kernel functionality from user space using Operating System abstractions (e.g. QLinux [30] or Dionisys [10]).

Hand-coding the kernel suffers from lack of flexibility. The obtained functionality is usually not reusable for different probing techniques. Also, using system abstractions in an environment running an operating system with quality of service guarantees is costly. These systems are also inherently complex.

While PERISCOPE is implemented in the kernel, it is not intended to be a tool to estimate a specific property. Instead, it is designed to be a *programmable* probing framework suitable for any inference technique that relies on active probing. By implementing a rich set of *basic* probing functions in the kernel, PERISCOPE can provide the necessary quality of service guarantees while avoiding the complexity of general purpose frameworks. By providing a generic user space API, PERISCOPE provides users with the flexibility to define and activate arbitrary probing structures.

III. Probing Terminology

In this section, we describe the basic terminology for the various probing sequences implemented in PERISCOPE, and which we use throughout the rest of the paper.

For the purposes of this paper, a *probe* is a sequence of one or more packets transmitted from a common origin. We say that any contiguous subsequence of packets within a probe are transmitted *back-to-back* if there is no time separation between transmission of the individual packets within the subsequence. A *multi-destination* probe is one in which the constituent packets of the probe do not all target the same destination IP address. Multi-destination probes have begun to see wider use as emulations of notional multicast packets—many of the same end-to-end inferences that can be made with multicast packets can be made with multi-destination unicast probes (albeit with added complexity) [13], [14]. A *uniform* probe is one in which all of the constituent packets are of the same size; likewise, a *non-uniform* probe consists of packets of different sizes. Finally, we say that an individual packet is *hop-limited* if its TTL is set to an artificially small value so as not to reach the ostensible destination. Hop-limited packets can be used to trigger an ICMP response from an intermediate router.

Throughout the paper we use various probing techniques that rely on sending sequences of probes. The probing techniques differ in the number of packets constituting a probe, the size, and the path traversed by each probe packet. They also differ in the host collecting the probing responses and the function used by this host to perform the required estimation.

In many instances, the process of probing may involve sending a multitude of probe structures to various destinations. For example, consider a multi-destination probe which targets two clients (from a single source). Given a set of clients, it may be necessary to send such a probe to every possible pair of clients in the set. We call such a set of probes a probing *round*.

Each packet p transmitted within a probe is parameterized by its size $s(p)$ in bytes and its final destination, $D(p)$. In the event that a packet is hop-limited, it has a third parameter, its maximum hop-count, $h(p)$. To denote a probe, we refer to each probe packet with a distinct lowercase letter, and represent the sequential order in which they are transmitted from the probing host by writing them from left to right. We denote interpacket spacing with square braces. As an example, $[pq][pq][r]$ would denote transmission of a pair of identical two-packet probes followed by a single packet probe which has different characteristics.

Finally, we use the term *interarrival time of packets p and q at a link* to denote the time elapsed between the arrival of the last byte of p and the arrival of the last byte of q at that link.

IV. Design Rationale

The design of PERISCOPE was driven by a number of objectives aiming to: (1) Minimize user/kernel boundary crossings. (2) Provide enough primitives to enable the definition of arbitrary probing structures and probing techniques. (3) Provide a structured and well-defined interface for applications. (4) Ensure kernel code modularity and restrict changes to the networking stack.

By implementing the scheduling and monitoring functionalities in the kernel, PERISCOPE minimizes user/kernel boundary crossings. The user/kernel boundary is crossed only during initialization operations[2] and during periodic application callbacks to report inference results. This optimization is valuable for busy servers.

PERISCOPE is designed to be general and friendly enough in the sense that users can define from user space the structure of their intended probes and various probing characteristics (e.g. probe packets inter-departure times, the characteristics of a probing round, the number of rounds needed, *etc.*). Also, although some versions of inference and labeling algorithms such as Bayesian Probing [14] and Cartouche Probing [15] are provided inside the kernel, a user can choose to get from the kernel some native results (e.g. arrival times of the responses of all probe packets in a round of probing) and implement its own inference algorithm in user space.

The interface between applications and PERISCOPE is done through the use of control sockets. System calls are translated through `ioctl` calls to perform appropriate actions in the kernel. An application uses the `select()` system call to receive PERISCOPE callbacks. This approach (control socket + select + ioctl)

[2] As we will detail later, this includes group setup and flow registration.

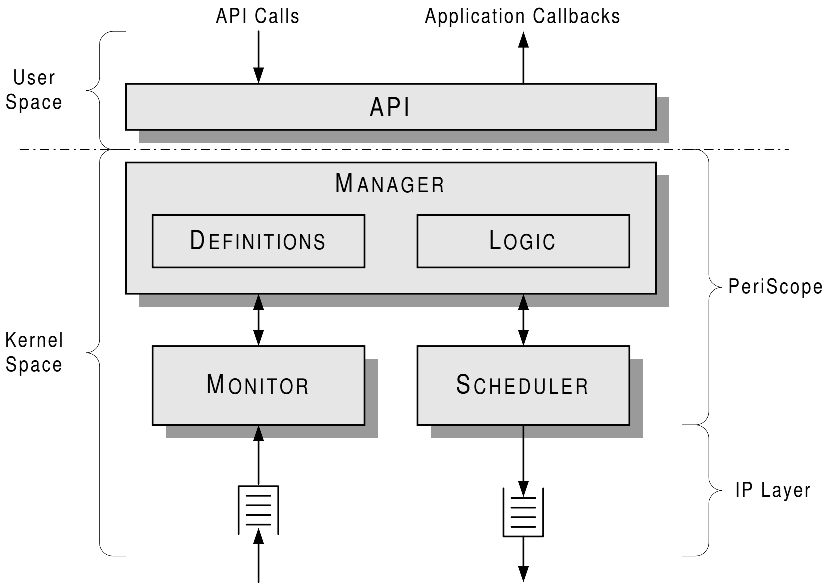


Fig. 1. PERISCOPE Architecture

restricts code changes caused by PERISCOPE to the networking stack and provides a well-defined and flexible interface for applications.

V. PERISCOPE ARCHITECTURE

PERISCOPE server-side functionality allows for: (1) the maintenance of the definitions of probing structures, (2) the orchestration of probe transmissions, (3) the collection of *relevant* probe packet responses (e.g., accurate arrival times, loss rates), and (4) the execution of the inference processes.

Figure 1 depicts the main components of the PERISCOPE architecture.

The `Manager` keeps record of all endpoints (i.e., clients) under consideration. Endpoints are partitioned into application-defined *groups*—groups of clients specified by applications through an API call. The inference procedures are applied on flows destined to endpoints belonging to a single group. This partitioning of flows into groups is designed mainly to give the flexibility of running more than one probing structure or running the same probing structure between different set of endpoints simultaneously. This may be useful for probing techniques that use more than one probing structure [14] or to diagnose shared congestion between different sets of endpoints possibly for aggregate congestion control purposes [3]. The `Manager` associates with each group a probing structure definition that contains the number of probe packets, the size and the destination (or the number of hops) of each probe packet. It also associates with each group a set of probing parameters (e.g., the number of probes to send and the probing rate). These definitions and parameters are provided by the applications in a generic way through API calls.

The `Scheduler` uses a timer for each group[3]. Whenever a timer associated with a group of endpoints expires, a new probe for this group is inserted into the IP stack for transmission. The inserted probe structure conforms to the probe definition associated with the group. The `Scheduler` ensures that the probe packets are inserted back-to-back on the top of the IP stack. In order to send packets back-to-back we extended the kernel library with a function that sends packets back-to-back with a single system call.

The `Monitor` keeps track of the relevant characteristics (e.g., losses and arrival times) of the probe packets responses. These statistics are updated as a result of the receipt of an ECHO REPLY or an ICMP TIME EXCEEDED message. A user may elect to receive the replies in user space by itself and store the response characteristics on its own.

VI. PERISCOPE API

In this section, we present some details of PERISCOPE's API[4].

As we explained earlier, the PERISCOPE API frees

[3]Note that the timer expiration time depends on the probing rate parameter associated with the group's probing structure definition

[4]For more details about PERISCOPE functionality refer to *http://cs-people.bu.edu/harfoush/periscope*

an application from having to manage the probing processes it requires, moreover it guarantees specific properties that are not possible to ensure from user space concerning the transmission of probe packets from an end-host (e.g., guarranteeing that probe packets are transmitted back-to-back or with a specific interdeparture time).

Using the PERISCOPE API, an application can define and activate a probing sequence by following the steps below:

1. Create a new *group* and provide the parameters associated with the group (e.g., the length of the probing sequence, probing rate, frequency of callbacks, sensitivity constant to be used for inference, etc.),

2. Register the endpoints (i.e., set of clients) to be considered as part of this group,

3. Define the probing structure (e.g. the number of probe packets in a probe and for each probe packet the flow that it belongs to, its TTL field value[5] and its size),

4. Activate the group, and

5. Wait for feedback.

Figure 2 shows the user level data structures and API used by PERISCOPE. We discuss these next.

Data Structures: The `group_param_t` structure contains the different parameters that define the settings for a specific group. A *phase* is defined as a transmission of one round of probes. The `max_phase` and the `probing_rate` fields of the `param_t` structure define the total number of probes and the probing rate that the application wants PERISCOPE to use. Based on the `feedback_type` value the user may elect to either receive probe packet replies by itself (`feedback_type`=0) or use the PERISCOPE feedback mechanism (`feedback_type`=1) and in this case the user can fix the feedback frequency through the `feedback_rate` field. The `group_param_t` structure include other group settings. The `flow_param_t` structure contains parameters relevant to a specific flow within a group. These include an identifier of the group to which the flow belongs and structures that define the flow destination.

PERISCOPE probe packets are transmitted as ECHO REQUESTs with a PERISCOPE header in its payload. The `payload_t` structure defines the sructure of the header to be included. A packet payload is timestamped by the kernel when it is posted in the IP stack for transmission (`probing_time`) and ECHO REPLY messages are timestamped when they reach the server (`feedback_time`). PERISCOPE manages two sequence numbers that are also included in the payload of the probe packets. These sequence numbers reflect the phase number that this probe packet belongs to (`phase_no`) and the probe packet position within the probe (`probe_packet_no`). The sequence numbers are mandatory for the application to compile probe packet replies.

```
typedef struct group_param {
  int max_phase;
  int probing_rate;
  int feedback_type;
  int feedback_rate;
  int cartouche_dimension;
  .
  .
  .
} group_param_t;

typedef struct payload {
  int group_id;
  int phase_no;
  int probe_packet_no;
  struct timeval probing_time;
  struct timeval feedback_time;
} payload_t;

typedef struct probe_packet {
  int flow_id;
  int size;
  int TTL;
} probe_packet_t;

typedef struct probe_structure {
  int n_probe_packets;
  probe_packet_t *probe_packets;
} probe_structure_t;

int PS_open(void);
int PS_new_group(int ctrl_fd,param_t params);
int PS_register_flow(int ctrl_fd,
                     int group_id,u_char *host);
int PS_register_probe_structure(int ctrl_fd,
    int group_id,probe_structure_t p_structure);
int PS_activate_group(int ctrl_fd,int group_id);
```

Fig. 2. PERISCOPE Data Structures and User Level API

In order to specify a probing construct, PERISCOPE defines two structures: `probe_packet_t` and `probe_structure_t`. The former defines the attributes of a probe packet (the group and flow to which it belongs, the size and the TTL field value to be assigned to this probe packet). The second structure has a list of all the probe packet definitions.

API: `PS_open()` creates a new socket of type SOCK_PS (a PERISCOPE-defined type used for the transmission and receipt of probes) and of protocol IPPROTO_ICMP (the ICMP protocol). The return value is socket file descriptor `ctrl_fd`. `PS_new_group()` allo-

[5]We set the TTL field value to 255 for probe packets that we want delivered to an endpoint.

cates a new group under `ctrl_fd`. The *parameters* of this group are passed along as arguments. The return value from this function is a group identifier `group_id`. `PS_register_flow()` registers a new endpoint with group `group_id` associated with the `ctrl_fd` socket. The user should call `PS_register_flow()` as many times as there are flows to be associated with the group. `PS_register_probe_structure()` is used to pass a probe structure definition to PERISCOPE `manager` in the kernel and `PS_activate_group()` activates `group_id`'s probing timer, thus starting the periodic probe transmission and statistics collection procedures for the group.

Probing Feedback: After activating a group, the user waits to receive probe responses. The responses can be either ECHO REPLY packets or ICMP TIME EXCEEDED packets. The former response is received from an endhost replying to ECHO REQUEST probe packet and the latter is received from a router on the way to the endhost after realizing that a packet TTL value reached 0 (as a result of a hop-limited probe packet). The user knows in advance, based on the TTL field value that he/she defined in the `probe_packet_t` structure whether a probe packet will trigger an ECHO REPLY or an ICMP TIME EXCEEDED message. The most appropriate way for a user to wait for all probe responses is to use a `select()` statement with its reading set of socket file descriptors containing only the `ctrl_fd` value returned by the `PS_open()` API call. After receiving a response packet, the user should classify it based on the `phase_no` and `probe_packet_no` fields in its payload to associate the response with the correct probe packet.

Sample Periscope Code: Figure 3 shows a sample routine, `Packet_Pair()`, detailing how we can program the uniform packet-pair (PP) technique to infer the end-to-end bottleneck bandwidth using PERISCOPE API. The PP technique has, using the terminology of section III, the [*pp*] probing structure. The parameters to the `Packet_Pair()` routine are the endpoint $D(p)$, the probing parameters to be used, the size of the probe packets and the timeout value to be waited before assuming that no more responses will be received.

Subtleties: When the response packet is ICMP TIME EXCEEDED, the received packet payload does not contain the `payload_t` structure as defined by the user and thus does not contain the expected `phase_no` and `probe_packet_no` fields. This might cause spurious

```
Packet_Pair(
                  u_char *host,
                  group_param_t params,
                  int size
                  struct timeval timeout;
            )
{

  int                ctrl_fd;
  int                group_id;
  int                flow_id;
  probe_packet_t     p_packet;
  probe_structure_t  p_structure;
  int                fd;
  fd_set             rset;
  ssize_t            r;

  ctrl_fd=PS_open();
  group_id=PS_new_group(ctrl_fd,params);
  flow_id=PS_register_flow(ctrl_fd,
                          group_id,
                          host);
  p_packet.flow_id=flow_id;
  p_packet.size=size;
  p_packet.TTL=255;
  p_structure.n_probe_packets=2;
  p_structure.probe_packets
         =malloc(2*sizeof(probe_packet_t));
  p_structure.probe_packets[0]=p_packet;
  p_structure.probe_packets[1]=p_packet;
  PS_register_probe_structure(ctrl_fd,
                             group_id,
                             p_structure);
  PS_activate_group(ctrl_fd,group_id);

  /* Wait for ECHO REPLY messages */
  FD_ZERO(&rset);
  FD_SET(ctrl_fd,&rset);
  for (;;) {
   if (
       (r=select(fd+1,&rset,NULL,NULL,&timeout))
        ==0
      )
      // timeout expired, so exit
   else
      // read payload details
  }
}
```

Fig. 3. Programming [*pp*] probing sequence of the packet-pair Probing technique using PERISCOPE API.

measurements due to mixing the responses especially when probes are lost or when packets can be reordered over the investigated path. To protect against this case, PERISCOPE provides two mechanisms:

1. Reordering Test: This test allows the determination of whether probe packets are prone to a change in their order over a path. This reordering can happen due to routers policies along the path. For example, in some probing experiments we conducted, we noticed that some routers give preference to mall packets over large packets. The test is based on sending all the probe packets to the end host (using a TTL value of 255 for all the packets) for a specific number of phases and checking that for each phase the returned ECHO REPLY messages have their assigned `probe_packet_no` field value in the same transmission order. If probe packets are prone to reordering the user is informed and has the decision as to whether to probe this path or to abort the probing process.

2. Synchronization Procedure: This procedure makes it simpler, in case that reordering does not happen, to identify the phase to which an ICMP TIME EXCEEDED reply belongs. Note that this procedure is useful if the first and last probe packet replies are expected to be ICMP TIME EXCEEDED messages. The test relies on sending a *synchronization packet* of small size and a TTL value of 255 after the transmission of each probe and using the ECHO REPLY messages triggered by these packets to separate the responses of different probe packets.

User Level Libraries: PERISCOPE is equipped with a set of user space libraries that implement the functionalities necessary to build and label *metric-induced network topologies* instantiated for the loss rate metric.

Recap: In this section, we have described the data structures and the user level API routines that can be used to program, activate and capture responses to generic probing structures using PERISCOPE. We have also pointed out some of the problems that might face an active probing tool in terms of packets reordering and packet loss and proposed precautions against these problems. In the next section, we turn our attention to validating PERISCOPE applicability in an Internet setting.

VII. Internet Measurement Experiments

In this section, we describe some of our experiments with Internet validation of PERISCOPE, drawing on examples taken from a large set of experiments that are aimed at using the PERISCOPE API to efficiently: (1) estimate the shared loss rate between a server and a set of clients and then infer and label *loss topologies*, and (2) estimate the bottleneck bandwidth along an arbitrary segment of a path.

The presented techniques are *efficient* in the sense that they do not need to probe every physical link along the shared path to determine the shared metric value as compared to the less efficient technique involving the use of *traceroute* [18] and *pchar* [24].

A. Loss Topology Inference

In this section, we describe an *illustrative* example demonstrating PERISCOPE's ability to correctly infer and label loss topologies in an Internet setting. In [4], a network topology connecting different endpoints

```
Bayesian_Probing(
                  u_char *host_A,
                  u_char *host_B,
                  group_param_t params,
                  int size
                  struct timeval timeout;
                 )
{

  int                ctrl_fd;
  int                group_id;
  int                flow_id_A;
  int                flow_id_B;
  probe_packet_t     p_packet_A;
  probe_packet_t     p_packet_B;
  probe_structure_t  p_structure;
  int                fd;
  fd_set             rset;
  ssize_t            r;

  ctrl_fd=PS_open();
  group_id=PS_new_group(ctrl_fd,params);
  flow_id_A=PS_register_flow(ctrl_fd,
                             group_id,
                             host_A);
  flow_id_B=PS_register_flow(ctrl_fd,
                             group_id,
                             host_B);
  p_packet_A.flow_id=flow_id_A;
  p_packet_A.size=size;
  p_packet_A.TTL=255;
  p_packet_B.flow_id=flow_id_B;
  p_packet_B.size=size;
  p_packet_B.TTL=255;
  p_structure.n_probe_packets=2;
  p_structure.probe_packets
         =malloc(2*sizeof(probe_packet_t));
  p_structure.probe_packets[0]=p_packet_A;
  p_structure.probe_packets[1]=p_packet_B;
  PS_register_probe_structure(ctrl_fd,
                                group_id,
                                p_structure);
  PS_activate_group(ctrl_fd,group_id);

  /* Wait for ECHO REPLY messages */
  FD_ZERO(&rset);
  FD_SET(ctrl_fd,&rset);
  for (;;) {
   if (
       (r=select(fd+1,&rset,NULL,NULL,&timeout))
       ==0
      )
      // timeout expired, so exit
   else
      // read payload details
  }
}
```

Fig. 4. Programming $[pq]$ probing sequence of the Bayesian Probing technique using the PERISCOPE API.

through IP routing can be represented in one of the following ways.

1. *Physical topology*, which has *all* routers connecting the endpoints represented as internal nodes of the topology,

2. *Logical topology*, in which all internal nodes with only one child have been collapsed into their parent recursively. As a result all internal nodes in a logical topology have at least two downstream endpoints,

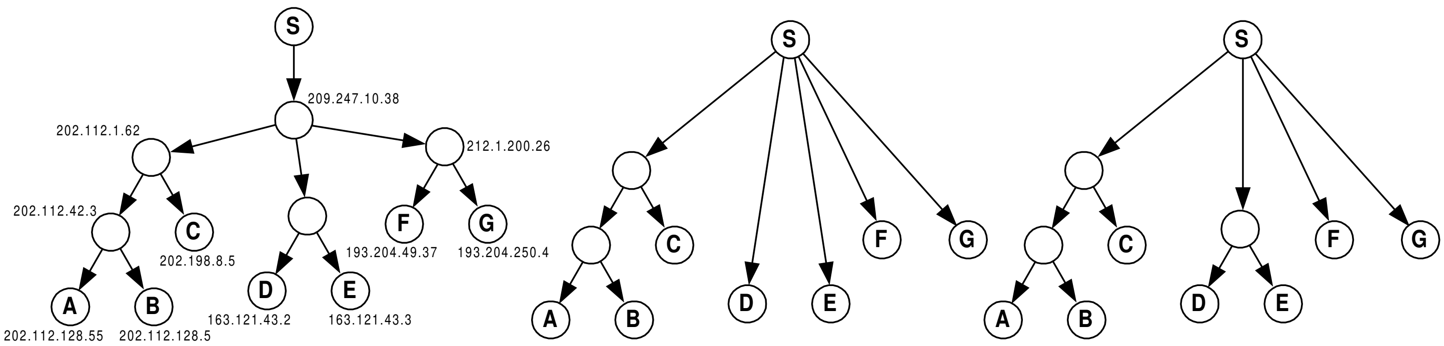


Fig. 5. PERISCOPE Validation: Logical tree used as a test case (left), most frequent inferred loss tree (middle) and minimal loss tree spanning all inferred trees (right).

3. *Metric-Induced topology* which is parametrized by a sensitivity parameter c and a metric of interest (e.g., loss rate), and is formed from the logical topology when all internal nodes with parent links of a metric value less than c have been collapsed into their parent. The larger the sensitivity constant the less links will appear in the tree as links with metric values less than the sensitivity constant are combined with other high metric value links. That is, the sensitivity constant is used to vary the resolution of the metric-induced topologies.

The technique used to infer the loss topologies relies on inferring the shared loss rate between each pair of clients and then using this information to build and label the loss topologies. In order to estimate the shared loss value PERISCOPE uses the BP technique [14] which we summarize next.

Bayesian Probing: Let S be a server connected through IP routing to two clients A and B. In order to estimate the loss rate over the shared path en route to clients A and B, the Bayesian probing technique requires that server S sends, using the terminology of section III, the following probing sequence: $[pq]$ where $s(p) = s(q)$, $D(p) = A$ and $D(q) = B$. Then by collecting probe responses and correlating their losses, the shared loss rate can be estimated. For more details about the bayesian probing technique refer to [14].

Periscope Code for BP Technique: Figure 4 shows a sample routine, `Bayesian_Probing()`, detailing how we can program the BP technique using the PERISCOPE API. The parameters to the `Bayesian_Probing()` routine are the endpoints A and B, the probing parameters to be used, the packet size $s(p)$ and the timeout value.

Experimental Setup: The topology used for the illustration connects a local server (Pentium II processor running RedHat Linux version 2.2.14) to a set of seven hand-picked hosts. The seven endpoints were selected to ensure the existence of different lossy paths that are shared between the server and various subsets of endpoints. In addition, by placing the server below a slow uplink, we ensured the existence of a (possibly) lossy path between the server and all endpoints. These choices were all made with the goal of stress-testing the inference and labeling techniques in mind.[6]

Figure 5 depicts the logical topology between the server and the seven hosts. Intermediate router IP addresses were obtained through the use of *traceroute*. The server is in the continental U.S. Hosts A,B and C are in China with hosts A and B on the same LAN of Beijing University of Aeronautics and Astronautics and host C in Northeast China Institute of Electric Power Engineering. Hosts D and E are in Egypt, on the same LAN of the Arab Academy for Science and Technology (AAST). Hosts F and G are in Italy at two different universities: Politecnico di Bari and Universita Degli-Studi di Bergano.

In order to study whether the results confirm to what the analysis and simulation in [4] expected, we need to establish a *reference* against which we could

[6]Specifically, validating our tool requires observing loss-topologies of appreciable structure—hence our choice of an inter-continental set of endpoints. Internet loss topology characterizations to small sets of random endpoints (from CAIDA/NLANR logs) rarely yielded rich/interesting structures. The depicted topology is meant to be illustrative, not representative. Also, experiments to the selected set of endpoints didn't consistently reveal high losses. Figure 5(right) was constructed only after integrating consistent inferred loss topologies viewed at different times.

compare the inferred and labeled loss trees we obtain for a given sensitivity parameter c. The logical tree (shown in Figure 5) is such a reference for $c = 0$. Obtaining such a reference for a non-zero sensitivity parameter is impossible since it requires knowledge of loss rates on all links of the logical tree. Moreover, loss rates cannot be assumed stationary for the duration of a PERISCOPE experiment and may not always be above the specified sensitivity parameter.

While the logical tree in Figure 5 cannot be used to directly validate loss trees inferred by PERISCOPE, it can be used to check whether the loss trees generated by PERISCOPE are *mutually consistent*. By mutually consistent, we mean that it is possible to construct the logical tree from the inferred loss tree by adding zero or more links.

We performed 20 experiments using PERISCOPE to infer and label the loss topology to the seven endpoints of the logical topology in Figure 5. These 20 experiments were conducted at different times. Each experiment consisted of 100 probing phases with 64-byte probes. At a probing rate of 5 probes/sec, it takes PERISCOPE about 4 minutes to complete 100 phases of probing. Notice that this time could be decreased by reducing the number of phases or by increasing the probing rate. Indeed, in most experiments, the loss topology tree stabilized within less than 10 phases—i.e. less than 24 seconds. However, increasing the probing rate is not desirable because it can lead to correlated probe packets behavior. This violates the BP approach assumptions.

Results: The non-stationarity of losses on the various links in a logical topology makes it unlikely that all of the potentially lossy links will be observable in a given experiment at a given time. Thus, one would expect that the loss topologies inferred by PERISCOPE will be different when run on the tree in Figure 5 (left). Indeed, PERISCOPE inferred six different loss topologies. Over the 20 experiments we conducted, the most frequently inferred loss topology tree is shown in Figure 5 (middle). This tree was inferred 11 times at times ranging between 3am and 7am EST (consistent with the fact that the lossy paths were the ones connecting our server to the hosts in China).

We also constructed the minimal loss tree [4] spanning all six of the loss topologies inferred by PERISCOPE when $c = 0.01$. The resulting tree (which itself is not one of the trees inferred by PERISCOPE) is shown in Figure 5 (right). Clearly, that tree is consistent with the logical tree depicted in Figure 5 (left).

B. Bottleneck Bandwidth of Arbitrary Path Segments

We use the term *bottleneck bandwidth* of a path to refer to the maximum transmission rate that could be achieved between two hosts at the endpoints of that path in the absence of any competing traffic. The bottleneck bandwidth of a path is limited by the minimum link speed along that path. In this section, we program PERISCOPE to infer the bootlenck bandwidth along an arbitrary path segment. In particular, given a path consisting of a sequence of links $L_1, \ldots, L_n$ with base bandwidths $b_1, \ldots, b_n$, our goal is to estimate the bottleneck bandwidth of an arbitrary sequence of links along that path, i.e. estimate $\min_{i \le k \le j} b_k$, for arbitrary i and j such that $i \le j \le n$. We use the shorthand $b_{i,j}$ to denote the bottleneck bandwidth in the interval between links i and j inclusive. The technique we use to do this inference is called *cartouche probing* [15] which we summarize next.

Cartouche Probing: Let L be a path consisting of a sequence of links $L_1, \ldots, L_n$ with base bandwidths $b_1, \ldots, b_n$. In order to estimate $b_{i,j}$, the Cartouche probing technique requires injecting at L_1 , using the terminology of section III, the following probing sequences: (1) $[pm\{pq\}^{r-1}pm]$ where $s(m) = s(q) \le s(p)$, $D(m) = n$ and $h(p) = h(q) = i$, (2) $[p_{i-1}m\{p_{i-1}q_{i-1}\}^{r-1}p_i m\{p_i q_i\}^{r-1}p_{i+1}m \ldots p_{j-1}m\{p_j q_j\}^{r-1}p_j m]$ where $s(p_w)$ and $s(q_w)$ are the same for all $w = i-1, \ldots j$, $s(m) = s(q_w)$ and $s(p_w) \ge s(m)$, $h(p_w) = w$, $h(q_w) = w$, $h(m) = n$. The technique then measure the inter-arrival times of the m probe packets and use it to infer b_i, j. For more details about the cartouche probing technique refer to [15].

Periscope Code for CP Technique: Figure 6 shows a sample routine, `Cartouche_Probing()`, detailing how we can program the $[pm\{pq\}^{r-1}pm]$ probing sequence of the CP technique using PERISCOPE API. where $s(m) = s(q) \le s(p)$, $D(m) = n$. The parameters to the `Cartouche_Probing()` routine are the probed endhost, the value i of $h(p) = h(q)$, the cartouche size r, the probing parameters to be used, the probe packet sizes $s(p) = s(q)$ (`large_size`) and $s(m)$ (`small_size`) and the timeout value.

Experimental Setup: We installed PERISCOPE on a laptop with a Pentium III processor running RedHat Linux 2.2.14. Four Internet paths connecting the laboratory to four different universities have been handpicked for the experiments. These universities are Georgia Tech in the US, University of British Columbia in Canada, Ecole Normale Superieure in France and Hirosaki University in Japan.

```
Cartouche_Probing(
                 u_char *host,
                 int i,
                 int r,
                 group_param_t params,
                 int large_size,
                 int small_size,
                 struct timeval timeout;
                )

{

  int               ctrl_fd;
  int               group_id;
  int               flow_id;
  probe_packet_t    p_packet_p;
  probe_packet_t    p_packet_q;
  probe_packet_t    p_packet_m;
  probe_structure_t p_structure;
  int               fd;
  fd_set            rset;
  ssize_t           r;

  ctrl_fd=PS_open();
  group_id=PS_new_group(ctrl_fd,params);
  flow_id=PS_register_flow(ctrl_fd,
                           group_id,
                           host);
  p_packet_p.flow_id=flow_id;
  p_packet_p.size=large_size;
  p_packet_p.TTL=i;
  p_packet_q.flow_id=flow_id;
  p_packet_q.size=small_size;
  p_packet_q.TTL=i;
  p_packet_m.flow_id=flow_id;
  p_packet_m.size=small_size;
  p_packet_m.TTL=255;
  p_structure.probe_packets
         =malloc((2*r+2)*sizeof(probe_packet_t));
  p_structure.probe_packets[0]=p_packet_p;
  p_structure.probe_packets[1]=p_packet_m;
  for (int index=1;index<r;index++) {
    p_structure.probe_packets[2*index]
                           =p_packet_p;
    p_structure.probe_packets[2*index+1]
                           =p_packet_q;
  }
  p_structure.probe_packets[2*r]=p_packet_p;
  p_structure.probe_packets[2*r+1]=p_packet_m;
  p_structure.n_probe_packets=2*r+2;
  PS_register_probe_structure(ctrl_fd,
                              group_id,
                              p_structure);
  PS_activate_group(ctrl_fd,group_id);

  /* Wait for ECHO REPLY messages */
  FD_ZERO(&rset);
  FD_SET(ctrl_fd,&rset);
  for (;;) {
   if (
      (r=select(fd+1,&rset,NULL,NULL,&timeout))
      ==0
     )
     // timeout expired, so exit
   else
     // read payload details
  }
}
```

Fig. 6. Programming the $[pm\{pq\}^{r-1}pm]$ probing sequence of the Cartouche Probing technique using PERISCOPE API.

In order to provide a preliminary study whether the results confirm what the analysis and simulation in [15] expected, we connected PERISCOPE to the Internet in two distinct locations: once through a 10Mbps LAN in Boston University Computer Science laboratory over and another time over a 56Kbps modem. Then by using the cartouche probing technique to estimate the bottleneck bandwidth over the first link, we know that the correct bandwidth estimates should be around 10Mbps and 56Kbps respectively.

In all experiments, we use $s(p) = 1500$ bytes and $s(m) = 60$ bytes. Experiments were conducted once per second until we obtain 100 valid results for a given path, recalling that packet reordering or packet losses invalidate an experiment.

Results: Figure 7 shows the histograms we obtained when using PERISCOPE to estimate the bandwidth estimate over the first link for the path connecting the laptop to the four universities both when connected over the 10Mbps with histogram bin width of 1Mbps (top) and over the 56Kbps modem with histogram bin width of 1Kbps (bottom row). In both cases the cartouche dimension r is 4 and the histogram used has a bin width of 1Mbps. These preliminary results are in keeping with our expectations.

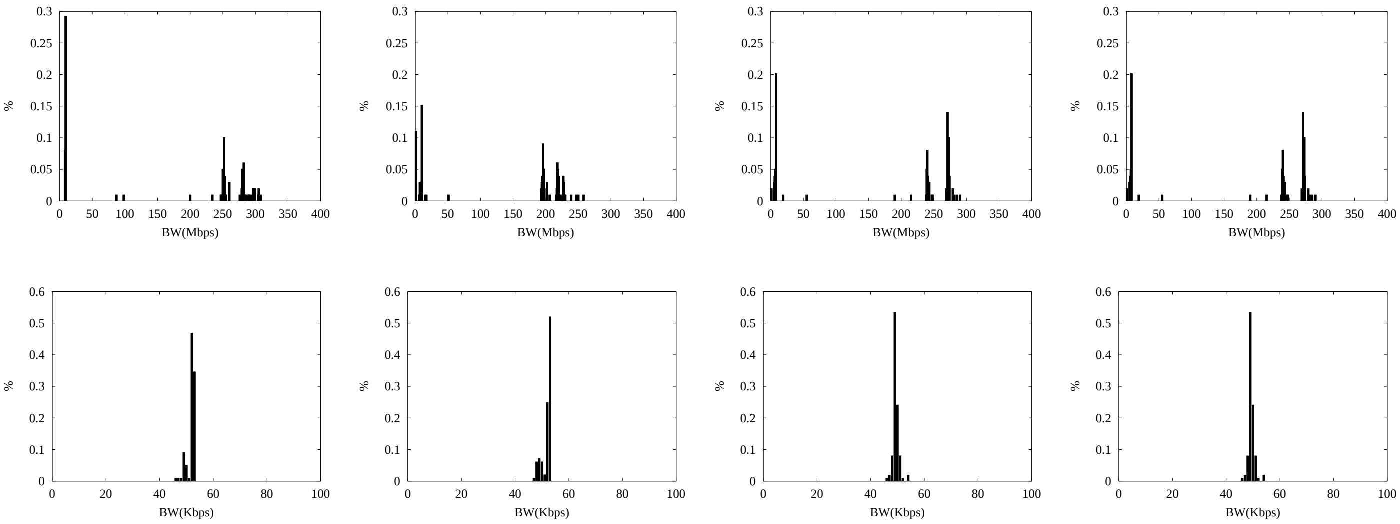
VIII. CONCLUSION

We have presented a Linux API called PERISCOPE that implements the functionality necessary to define, activate and collect measurements for arbitrary probing structures. PERISCOPE optimizes user/kernel boundary crossings while providing a flexible and well established interface to applications.

By presenting sample code that implements a variety of different probing techniques from the literature; namely the packet-pair technique, bayesian probing and cartouche probing; we have shown that PERISCOPE is easily programmable and is capable of generating different probing structures.

PERISCOPE has proven to be applicable in real Internet settings. We gave evidence of this applicability through two different sets of experiments to infer and label loss topologies and to infer the bottleneck bandwidth along arbitrary path segments.

Software Availability: PERISCOPE software is available for downloading at: *http://cs-people.bu.edu/harfoush/periscope*. The web page also includes more technical details, as well as a PERISCOPE tutorial.



Georgia Tech | University of British Columbia | Ecole Normale Superieure | Hirosaki University

Fig. 7. Histograms of bandwidth estimates over the first link connecting PERISCOPE to four universities over a 10Mbps with histogram bin width of 1Mbps (top) and over a 56Kbps modem with histogram bin width of 1Kbps (bottom). Cartouche dimension $r = 4$.

REFERENCES

[1] A. Barbir et al. Known CDN Request-Routing Mechanisms. http://www.globecom.net/ietf/draft/draft-cain-cdnp-known-request-routing-01.html, February 2001.

[2] D. Andersen, H. Balakrishnan, M. F. Kaashoek, and R. Morris. Resilient Overlay Networks. In *Proceedings of SOSP 2001*, Banff, Canada, October 2001.

[3] H. Balakrishnan, H. Rahul, and S. Seshan. An Integrated Congestion Management Architecture For Internet Hosts. In *SIGCOMM '99*, Cambridge, MA, September 1999.

[4] A. Bestavros, J. Byers, and K. Harfoush. Inference and Labeling of Metric-Induced Network Topologies. In *INFOCOM'02*, New York, NY, June 2002.

[5] J. C. Bolot. End-to-end Packet Delay and Loss Behavior in the Internet. In *SIGCOMM '93*, pages 289–298, September 1993.

[6] R. Cáceres, N. G. Duffield, S. B. Moon, and D. Towsley. Inference of Internal Loss Rates in the MBone. In *IEEE Global Internet (Globecom)*, Rio de Janeiro, Brazil, 1999.

[7] Robert L. Carter and Mark E. Crovella. Measuring bottleneck link speed in packet switched networks. In *PERFORMANCE '96, the International Conference on Performance Theory, Measurement and Evaluation of Computer and Communication Systems*, October 1996.

[8] Y.-H. Chu, S. Rao, and H. Zhang. A Case for End-System Multicast. In *ACM SIGMETRICS '00*, Santa Clara, CA, June 2000.

[9] M. E. Crovella, R. Frangioso, and M. Harchol-Balter. Connection Scheduling in Web Servers. In *Proceedings of 1999 USENIX Symposium on Internet Technologies and Systems (USITS '99)*, Boulder, CO, October 1999.

[10] `Dionisys`: End-System QoS Support for Distributed Applications. http://www.cs.bu.edu/fac/richwest/research.html.

[11] Constantinos Dovrolis. Pathrate: Measurement tool for the capacity and load of internet paths. http://www.cis.udel.edu/~dovrolis/bwmeter.html.

[12] Allen B. Downey. Clink: a tool for estimating internet link characteristics. http://rocky.wellesley.edu/downey/clink/.

[13] N. Duffield, F. Lo Presti, V. Paxson, and D. Towsley. Inferring Link Loss Using Striped Unicast Probes. In *IEEE INFOCOM 2001*, April 2001.

[14] K. Harfoush, A. Bestavros, and J. Byers. Robust Identification of Shared Losses Using End-to-End Unicast Probes. In *8th International Conference on Network Protocols (ICNP)*, Osaka, Japan, November 2000.

[15] Khaled Harfoush, Azer Bestavros, and John Byers. Measurement of Shared Bottleneck Bandwidth Using End-to-End Cartouche Probing. Technical Report BUCS-TR-2000-016, Boston University, Computer Science Department, July 2001.

[16] IPMA : Internet Performance Measurement and Analysis. http://www.merit.edu/ipma.

[17] V. Jacobson. `Pathchar`: A Tool to Infer Characteristics of Internet Paths. ftp://ftp.ee.lbl.gov/pathchar.

[18] V. Jacobson. traceroute. ftp://ftp.ee.lbl.gov/traceroute.tar.gz, 1989.

[19] J. Jannotti, D. Gifford, K. Johnson, M. F. Kaashoek, and Jr. J. O'Toole. Overcast: Reliable Multicasting with an Overlay Network. In *Proceedings of OSDI 2000*, San Diego, CA, October 2000.

[20] J. Kangasharju, J. Roberts, and K. W. Ross. Object Replication Strategies in Content Distribution Networks. In *Proceedings of WCW'01: Web Caching and Content Distribution Workshop*, Boston, MA, June 2001.

[21] S. Keshav. *Congestion Control in Computer Networks.* PhD thesis, University of California at Berkeley, September 1991.

[22] K. Lai and M. Baker. Measuring Link Bandwidths Using a Deterministic Model of Packet Delay. In *SIGCOMM '00*, Stockholm, August 2000.

[23] K. Lai and M. Baker. Nettimer: A tool for Measuring Bottleneck Link Bandwidth. In *Proceedings of USITS '01*, March 2001.

[24] B. Mah. `pchar`.

http://www.employees.org/~bmah/Software/pchar/, 2000.

[25] Mtrace: Tracing multicast path between a source and a receiver. http://www-itg.lbl.gov/mbone/mtrace.tips.html.

[26] V. Padmanabhan. Optimizing Data Dissemination and Transport in the Internet. Slides presented at the BU/NSF Workshop on Internet Measurement, Instrumentation and Characterization, September 1999.

[27] V. Paxson. End-to-end Routing Behavior in the Internet. In *SIGCOMM '96*, Stanford, California, August 1996.

[28] V. Paxson. End-to-end Internet Packet Dynamics. In *SIGCOMM*, 1997.

[29] V. Paxson. *Measurements and Analysis of End-to-end Internet Dynamics*. PhD thesis, U.C. Berkeley and Lawrence Berkeley Laboratory, 1997.

[30] `QLinux`: A QoS enhanced Linux Kernel for Multimedia Computing. http://lass.cs.umass.edu/software/qlinux/.

[31] P. Radoslavov, R. Govindan, and D. Estrin. Topology-Informed Internet Replica Placement. In *Proceedings of WCW'01: Web Caching and Content Distribution Workshop*, Boston, MA, June 2001.

[32] S. Ratnasamy and S. McCanne. Inference of multicast routing trees and bottleneck bandwidths using end-to-end measurements. In *Proceedings of IEEE INFOCOM '99*, pages 353–60, March 1999.

[33] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In *Proceedings of ACM SIGCOMM'01*, San Diego, CA, August 2001.

[34] Maya Yajnik, Sue Moon, Jim Kurose, and Don Towsley. Measurement and modelling of the temporal dependence in packet loss. In *Proceedings of IEEE INFOCOM '99*, pages 345–52, March 1999.