Design and Evaluation of the S^3 Monitor Network Measurement Service on GENI

Ethan Blanton*, Sarbajit Chatterjee†, Sriharsha Gangam*, Sumit Kala†, Deepti Sharma†, Sonia Fahmy*, Puneet Sharma†
*Department of Computer Science, Purdue University; †Hewlett-Packard
*{eblanton, fahmy, sgangam}@purdue.edu, †{sarbajit.chatterjee, sumit.kala, deeptis, puneet.sharma}@hp.com

***Abstract*—Network monitoring capabilities are critical for both network operators and networked applications. In the context of an experimental test facility, network measurement is important for researchers experimenting with new network architectures and applications, as well as operators of the test facility itself. The Global Environment for Network Innovations (GENI) is a sophisticated test facility comprised of multiple "control frameworks." In this paper, we describe the design and implementation of S^3 Monitor, a scalable and extensible monitoring service for GENI. A key feature of S^3 Monitor is a flexible design that allows easy "plug in" of new network measurement tools. We discuss our deployment experiences with S^3 Monitor on GENI, and give experimental results to quantify the performance and system footprint of S^3 Monitor. We find that the S^3 Monitor service is light-weight and scales well as the number of paths to be monitored increases.**

I. Introduction

End-to-end network measurements such as delay, loss, and available bandwidth are essential for network monitoring, evaluation of network architectures, and optimizing and managing network services. The importance of a planned measurement infrastructure that can be shared among multiple users and applications is well-recognized [1], [2], [3], [4], [5], [6], [7]. In the Global Environment for Network Innovations (GENI) [8], monitoring and measurement are critical to both infrastructure operators for fault localization and tuning, and for experimenters using GENI for their networking research. Sharing a network measurement service can significantly reduce measurement overhead, increase accuracy, and remove the burden of performing network measurements from individual users.

In this paper, we describe the design, implementation, and operational experience with a system, the Scalable Sensing Service (or S^3 Monitor for short), that provides a framework for users to take controlled *active measurements* between GENI [8] nodes. The system significantly extends our prototype service [9] deployed on PlanetLab [10], and ports it to the GENI environment with its multiple *control frameworks*. A key feature of our design is extensibility, allowing the user to utilize arbitrary measurement tools, as long as they are adequately described, and we give such a description mechanism. For example, to measure available bandwidth between two GENI nodes, the user may choose to use pathChirp [11], pathload [12], Spruce [13], or develop and use a new tool. Similarly, to measure bottleneck capacity, the user may select pathrate [14], CapProbe [15], or other approaches [16], [17], [18]. A second key feature is portability, as the resources available via GENI are heterogeneous. We describe our experiences in creating a portable yet flexible and small-footprint distributed measurement system using interpreted code with minimal run-time requirements.

978-1-4673-0298-2/12/$31.00

The S^3 Monitor service includes a web interface for users to schedule measurements across GENI nodes, manage ongoing measurements, and retrieve measurement results [19]. The service manages the dissemination of measurement schedules to GENI nodes, and the retrieval of measurement results from these GENI nodes on behalf of the user. Measurement results are stored in a repository by the system for later reference until purged by the user. A GENI experimenter can therefore deploy our service to a GENI experiment, and use its facilities to collect network measurements within the experiment "slices."

The remainder of this paper is organized as follows. Section II gives some background and outlines the goals of our system. Section III gives an overview of the system architecture. Section IV describes our extensible design and implementation that allow using new measurement tools. Section V quantifies the system performance and resource utilization using a set of benchmarks. Section VI discusses the lessons learned from our experiences with the system implementation and deployment. Section VII summarizes related work. Finally, Section VIII concludes the paper and discusses future work.

II. Background and Goals

Network measurement data is most useful if measurements can be requested by users on-demand at desired *times* and *frequencies*, using desired *measurement tools*. While several studies have researched or developed measurement infrastructures, *e.g.*, [1], [3], [4], our approach is guided by the observation that the utility of a measurement service is significantly increased by the *extensibility* of the queries it can answer, as well as its *graceful handling* of heavy measurement request load.

Previous work either does not allow on-demand requests, uses hard-coded measurement tools, or lets each user independently invoke measurement tools and then places static filters on their traffic. In contrast to these extremes, we have designed

and implemented an extensible yet safe service for handling on-demand measurement queries. Our S^3 Monitor service is tailored to GENI experimenters. Before discussing our design goals, we present a brief overview of the GENI infrastructure to introduce the terminology used throughout the paper.

A. GENI: The Global Environment for Network Innovations

GENI is a National Science Foundation initiative for creating a *Global Environment for Network Innovations* to foster research and experimentation with emerging network architectures. GENI provides a collaborative and exploratory infrastructure to network researchers, also referred to as *experimenters*, for performance evaluation and correctness validation of their proposed network architectures and systems. GENI experimenters can request and create virtualized custom network environments, called *slices*, consisting of programmable hosts from different *clusters*, also known as *control frameworks* or *aggregates*. Current aggregates include *PlanetLab* [10], *ProtoGENI* (with nodes in Utah and Kentucky) [20], and *Million Node GENI* [21]. The GENI infrastructure can connect the end-host resources using a programmable network within the cluster, or federate across multiple clusters over a wide-area backbone network.

The resources for a *slice* (including the hosts and the connectivity among them) are requested using GENI's resource specification language, an XML format called *RSpec* [22]. The user can describe his/her own configuration of resources in RSpec, and use it to request a GENI slice. In response to the slice request, the user receives a *manifest* (following a related RSpec schema) that contains slice-specific information about the allocated resources (*e.g.,* dynamically assigned IP addresses, host names, and node names). There are different services available in the GENI framework to allocate network resources described in the RSpec. All these services also provide a manifest in response, describing the allocated resources.

B. Design Goals

The deployment of our measurement service on GENI clusters and the need to make the service available to experimenters imposed several constraints on the design. There were three key requirements from prospective users of the S^3 Monitor service:

- **Portability:** The GENI infrastructure has multiple *clusters*. Since the experimenters need flexibility to test out their research ideas on different clusters, we designed our S^3 Monitor service to be portable across clusters. Even within each cluster, our service had to be portable across a variety of hardware and software platforms (*e.g.,* operating systems, network connectivity).
- **Extensibility:** As GENI experimenters develop and test new network protocols and services on the GENI clusters, they may also need to develop novel measurement and monitoring tools. Hence, all components of S^3 Monitor service should provide an extensible framework and APIs to allow the ability to plug-in, control, collect, and query data from these new tools. There should be no requirement that the tools are written with S^3 Monitor integration as a design goal. Instead, S^3 Monitor should have the capability of utilizing third-party tools directly.
- **Management and experiment plane separation:** The GENI clusters allow an experimenter to request networked resources with particular features and network connectivity. During the course of an experiment, S^3 Monitor measures the network properties of the experimental plane. GENI experimenters often vary the testbed/slice network, sometimes even disconnecting some nodes in the topology, to evaluate their research. Hence, S^3 Monitor is designed to support a separate management plane that maintains connectivity with experimental nodes to control the measurements.

C. Design Decisions

Based on the above requirements, we made the following design decisions for architecting the S^3 Monitor service. These also reflect the enhancements we had to make to our earlier prototype [9] for GENI deployment:

- **Support GENI RSpec:** The GENI control framework teams have been working on the definition of a resource specification called RSpec [22] that can be used by experimenters to allocate resources for their experiments. The S^3 Monitor service takes the manifest file generated by the resource allocation process as input to collect meta-data about the measurement setup for the experiment. This meta-data will be used to control and configure measurement requests.
  Fig. 1 depicts the interaction between a GENI experimenter and the S^3 Monitor service during experiment setup.

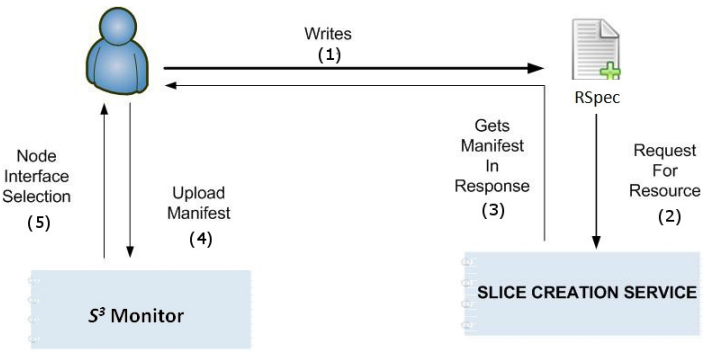

Fig. 1. Experiment Setup with S^3 Monitor.

- **Sensor pod portability and extensibility:** The portability requirement described above is most critical to the implementation of the sensor pod. The sensor pod service was re-implemented in interpreted Python with minimal external run-time requirements to support a wide variety of platforms. Further, in order to make the sensor pod as extensible as possible, new sensors may be bound to the pod by experimenters at run-time.
- **RSpec-based sensor deployment:** GENI RSpecs define mechanisms for installing software on GENI nodes

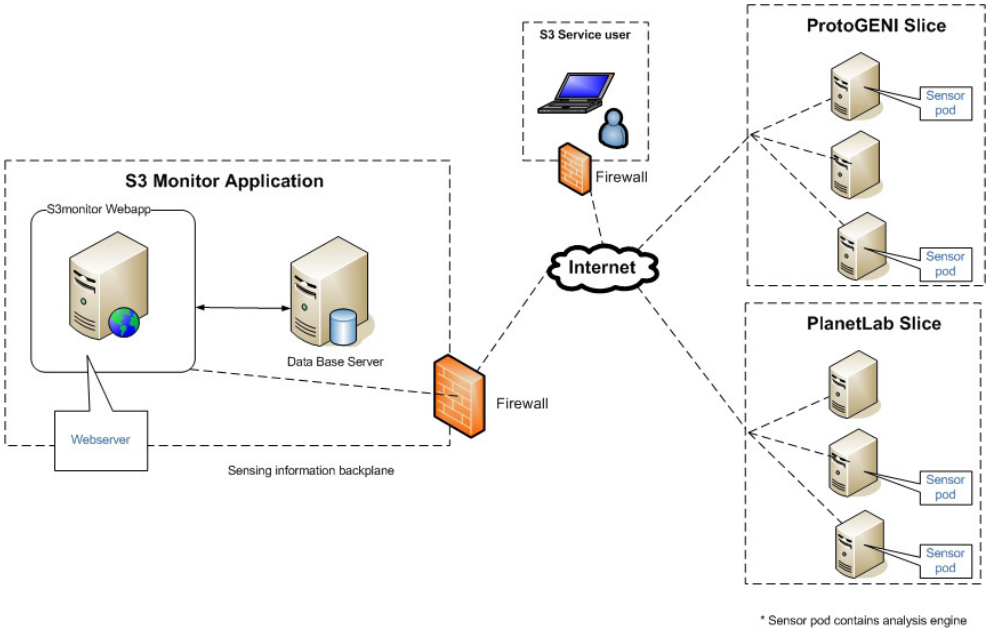

Fig. 2. S^3 Monitor Architecture

during experiment initialization. We provide support for installing the sensor pod using this interface.

- **Web services-based data acquisition:** The separation requirement between the management and experimental planes entailed a design where a web portal is provided to the experimenter, and the portal interfaces with a management server that interfaces with the GENI nodes using a service-oriented architecture, as discussed in the next section.

III. System Architecture

According to our design goals and strategies discussed above, S^3 Monitor is divided into two primary components (depicted in Fig. 2):

1) *Sensor pods*: A sensor pod is a light-weight web service-enabled framework which hosts *pluggable* measurement sensors (*e.g.*, ping, pathChirp [11], tulip [23]). The sensor pod executes on each node that serves as an end point (source or sink) of a measurement. The sensor pods can be deployed on GENI nodes, where a GENI node is a reserved node used by a GENI experimenter.
2) *Sensing information manager*: The sensing information manager is the application with which the user interacts. The manager triggers measurements on the sensor pods and collects measurement data. It includes a web application (portal), written in Java. The sensing information manager can be installed on any host which can establish communication with the GENI nodes hosting sensor pods.

A. Sensor Pod

Fig. 3 depicts the components of the sensor pod. The sensor pod processes requests from the sensing information manager via a built-in web server. The services on the sensor pod are written in Python, and contain a framework for plugging-in sensors. Sensors extend the interface provided by S^3 Monitor to manage measurement tools. Each sensor is responsible for execution of its measurement tool, data collection, and clean-up.

The sensor pod includes the following modules:

- *Sensors* are invoked by S^3 Monitor to measure properties such as latency, loss, bottleneck capacity, and available bandwidth between two nodes. Each sensor provides three functions. It performs the measurement requested by the user, processes and returns the measurement results, and frees up any measurement-related resources when it completes.
- The *Sensor Interface* provides the framework for supporting run-time binding of pluggable sensors. The measurement results collected by the sensors are relayed back to the sensing information manager via the *Python CGI module*.
- The *Python CGI module* provides the back-end web service to process measurement requests invoked by the sensing information manager. It includes scheduling and invocation services for taking *instant* (non-scheduled measurements requested by the user in real-time) and *periodic* (repeating measurements scheduled in advance) measurements.
- The *Data Repository* has measurement data collected by the Python CGI module during periodic measurements for later retrieval.

B. Sensing Information Manager

The sensing information manager is a web portal that provides management facilities to control the sensor pods and

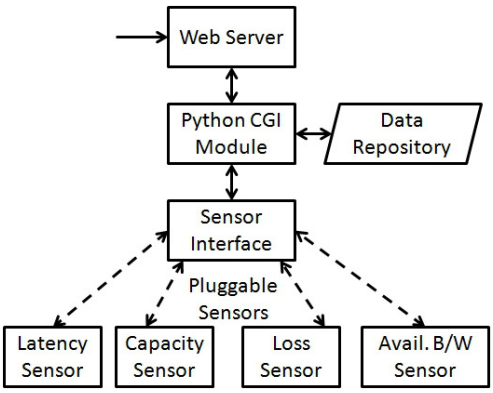

Fig. 3. Sensor Pod Architecture

maintain measurement data. Fig. 4 depicts the components of the manager. The manager is implemented as a collection of Java servlets running under Apache Tomcat. These servlets provide a user interface for binding sensor pods to S^3 Monitor, scheduling measurements, and retrieving measurement results. The complete web application combines static HTML content, the Java servlets, and client-side JavaScript to offer complete control of S^3 Monitor. The S^3 *Monitor Application* component processes the user queries and commands from the web application, and interacts with various sensor pods to invoke measurements and collect results.

As mentioned above, the sensing information manager supports two methods for sensor interaction. Users may request either *instant* or *periodic* measurements. Instant measurements invoke sensors immediately and return their results via the web application in real-time. Periodic measurements are scheduled for future (possibly repeating) invocation, and their results are collected for later viewing. Fig. 5 depicts a sample screen shot of the user interface for querying and displaying periodic measurement results.

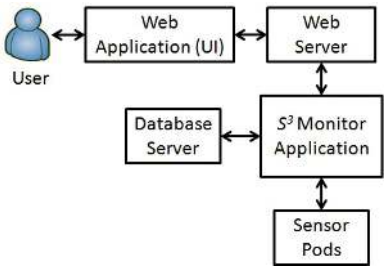

Fig. 4. Sensing Information Manager Architecture

IV. Portable and Extensible Sensor Pods

As described in Section III, the S^3 Monitor sensor pod provides a framework for hosting S^3 Monitor sensors and communicating with the sensing information manager. Recall that a sensor is a piece of software which collects information about some network property, and that the sensor pod software runs on every host which supplies sensing information to the S^3 Monitor service.

Fig. 5. Sample screen shot showing the interface to display periodic measurement results

In our previous prototype service [9], the sensor pod was a monolithic piece of software, with the individual sensor management routines embedded in the sensor pod itself. In that architecture, the sensor pod must be modified in order to add or remove sensors, meaning that modifying the sensor capabilities of a running pod requires editing the sensor pod source and re-deploying a new version of the sensor pod. In the new S^3 Monitor sensor pod, the sensor pod offers a *pluggable* sensor interface for which individual sensor management plugins (hereafter simply *sensors*) may be supplied. A sensor consists of a sensor description in XML and a Python module implementing the S^3 Monitor sensor interface which performs the actual sensing (or invokes an external program to do so).

Our previous prototype service [9] also made use of native compiled code, which restricted each sensor pod build to usage on a single platform. Because the sensor pod software must execute on every host which provides sensor information, this means that either the sensing hosts must be largely homogeneous in nature, or a variety of sensor pod images must be managed and deployed to the appropriate hosts. The new S^3 Monitor sensor pod is implemented entirely in platform-independent interpreted code, enabling a single sensor pod build to run on a variety of platforms and greatly simplifying deployment.

A. Describing Sensors

Sensor plugins consist of two logical components: a sensor description in XML, and a Python module implementing the sensor logic. Sensor plugins are dynamically loaded by the sensor pod at run time, and may be provided either as part of the pod deployment or by the experimenter. Several standard sensors are provided with the pod, such as ping for delay sensing and traceroute for end-to-end path sensing.

The sensor description is a chunk of XML data which describes the sensor input parameters, output metrics, and other relevant metadata such as the name of the sensor. Fig. 6 shows an abridged sensor description schema. The sensor description is used by both the sensor pod and the sensing information manager to describe the capabilities and

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

  <xs:element name="name" type="xs:string" />
  <xs:element name="description" type="xs:string" />
  <xs:element name="module_name" type="xs:string" />

  <xs:element name="parameter">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" minOccurs="1" maxOccurs="1" />
        <xs:element ref="description" minOccurs="0" />
        <xs:element ref="type" minOccurs="1" maxOccurs="1" />
        <xs:element ref="unit" minOccurs="0" />
        <xs:element ref="default_value" minOccurs="0" />
        <xs:element ref="max_value" minOccurs="0" />
        <xs:element ref="min_value" minOccurs="0" />
        <xs:element ref="regex" minOccurs="0" />
        <xs:element ref="required" minOccurs="0" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

  <xs:element name="metric">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="name" />
        <xs:element ref="description" />
        <xs:element ref="type" />
        <xs:element ref="unit" />
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Fig. 6. Abridged Sensor XML Description Schema

```
<?xml version="1.0" encoding="UTF-8" ?>
<sensor>
  <name>ping</name>
  <description>Probes the destination with ICMP Echo Requests</description>
  <module_name>ping.py</module_name>
  <parameters>
    <parameter>
      <name>count</name>
      <description>Number of packets to send</description>
      <type>integer</type>
      <default_value>5</default_value>
      <required>false</required>
    </parameter>
    <parameter>
      <name>interval</name>
      <description>Delay between packets</description>
      <type>integer</type>
      <unit>sec</unit>
      <required>false</required>
    </parameter>
  </parameters>
  <provides>
    <metric>
      <name>latency</name>
      <description>Measured round-trip latency to destination</description>
      <type>integer</type>
      <unit>msec</unit>
    </metric>
    <metric>
      <name>loss</name>
      <description>Proportion of probes (or replies) lost</description>
      <type>float</type>
      <unit></unit>
    </metric>
  </provides>
</sensor>
```

Fig. 7. Basic Ping Sensor Description

configuration of a sensor, so in addition to input parameter names and output parameter value formats, it contains human-readable descriptions which are presented to the experimenter via the web interface.

Sensor inputs, or *parameters*, values which modify the sensor behavior, are defined by a name used to identify the parameter and type (integer, string, etc.), as well as optional properties which further specify their usage. These optional values include a human-readable description, the minimum and maximum values and unit in which numeric parameters are measured (for example, ms or kB), the default value to be used if none is provided by the user, and a regular expression which can be used to validate the user-provided value. There is also an indicator which specifies whether the parameter must be provided by the user, or is optional.

Sensor outputs, or *metrics*, are also defined by a name and type as well as their unit and a human-readable description. A sensor may provide zero or more metrics of varying type and unit. Some "well-known" metrics are defined, such as latency and loss, but sensors may provide metrics of any kind. By providing for extensible output metrics, users are afforded the opportunity to use S^3 Monitor to manage new and experimental sensors.

Sensor logic is provided by a Python module which is likewise loaded at run time by the sensor pod. This module executes the sensor as an external task according to parameters defined by the sensor description, then processes its results and returns those metrics defined in the sensor description.

Fig. 7 shows a sample sensor description for the common *ping* tool. The first few lines describe meta-information about the sensor, such as its identifying name (ping) and a human-readable description, as well as the name of the Python source file which implements its logic. Its parameters (count, interval) and metrics (latency, loss) are listed following this, along with their type and units.

B. Portability

Deploying the S^3 Monitor sensor pod to GENI hosts presents a more challenging problem than our prototype PlanetLab deployment [9] due to portability concerns. The GENI resources allocated by an experimenter can take the form of almost any piece of hardware imaginable, from smart switches to routers to PCs to wireless access points, and in many cases the operating system or software platform running on the resource is configurable by the experimenter. The scope of resources targeted for S^3 Monitor sensor pod deployment is somewhat narrower, consisting of PC-like machines running a UNIX-like operating system. However, within this scope there are dozens of operating system images available on several different hardware platforms.

Our prototype deployment on PlanetLab ran on largely homogeneous resources, in the form of Intel 32-bit x86 PCs running a Red Hat 9 operating system image. Our initial deployment on GENI made similar assumptions, and consequently included a native-code web server as well as several native-code sensors bundled as binary executables. However, supporting the broad set of target platforms available on GENI makes precompiled native code a problematic deployment method. Therefore, the sensor pod was modified to provide all of its basic functionality in Python [24], reducing its platform requirements to a working Python 2.5 interpreter and a Bourne-compatible shell (for installation and startup).

To accomplish this, two major changes were made. First, the native code web server was replaced with a pure Python web server implemented using the Twisted.web [25] networking engine. Second, Twisted.web itself and other Python code dependencies were reduced to pure Python code, with any optional and/or unnecessary native code removed, and these

pure Python support libraries bundled with the sensor pod software.

This platform-independent implementation, coupled with a platform-independent installation and invocation process, allows for simple sensor pod deployment to GENI slices. The S^3 Monitor sensor pod can be directly deployed to hosts allocated on one GENI aggregate (ProtoGENI) in a fully automated fashion, by simply including a few lines in the RSpec resource specification which describes the GENI slice. For other aggregates completely automated deployment is an ongoing project, but in the typical case manual deployment requires only transferring and extracting one archive, and executing one command.

C. Structured Output

Sensor descriptions (described in Section IV-A) define output metrics in terms of values of a defined type and units, while the output of many network measurement tools is more or less ill-defined text. This text may be provided directly to the experimenter using a unitless string-type metric, but in many cases it may be appropriate to provide more structured output. For example, the *ping* sensor described in Fig. 7 provides the measured latency to the destination host in milliseconds, as well as the measured probe loss rate as a unitless proportion.

This structured output serves two purposes. First, the text output of the "same" tool may vary across multiple platforms. To continue with our *ping* example, there are numerous implementations of the *ping* tool which provide essentially the same information with (sometimes subtly) different output formats. Extracting the salient information for the specific platform on which the sensor pod is running removes this burden from the experimenter. Second, extracting the values of well-known metrics with specified units allows specific sensors to be somewhat fungible, as long as they provide information about the metric of interest.

There is ongoing work within the GENI Instrumentation and Measurement working group [26] to define a network measurement ontology and a schema for specific metrics, as well as to provide measurement archival services. Reducing the output of measurement tools to structured output facilitates inter-operation with such a system, both simplifying and encouraging the sharing of measurement data.

V. System Performance and Footprint

S^3 Monitor accepts measurement requests, schedules measurements, invokes measurement tools, and stores measurement results. Each of these operations consumes network bandwidth, disk I/O bandwidth and storage, main memory, and CPU time across the deployment nodes. To evaluate the impact of S^3 Monitor on these resources, we mimic user behavior with a web browser emulator and monitor the effect on system resources. We use the collectl [27] tool installed on GENI nodes to collect CPU, memory, and disk utilization. Collectl extracts periodic snapshots of a variety of system properties. Even with a default monitoring frequency of 1 second, the overhead of collectl itself is very low (less than 0.1% on most

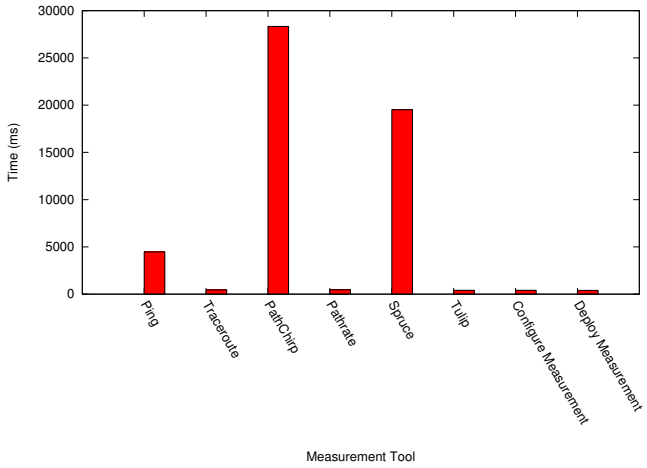

Fig. 8. Response times of different measurement tools and operations on S^3 Monitor.

systems) [28]. We use the Selenium web driver [29] to emulate user measurement queries.

A. Response Time

The response time of a measurement request is defined as the time elapsed between a user's measurement request and the return of measurement results back to the user. Fig. 8 depicts the average response time of different measurement tools invoked using the S^3 Monitor service deployed over ProtoGENI. We note that the pathChirp tool [11] has the maximum response time of about 30 seconds. The pathrate tool [14] can also take significant time but we had utilized its "quick terminate" mode in this experiment. Most measurement requests take less than 300 ms. The steps of configuring and deploying periodic measurements take negligible time, indicating scalability of S^3 Monitor. The parameters used when instantiating the measurement tools are listed in Table I.

B. System Footprint

Fig. 10 gives results from a 15-minute periodic ping experiment on a small deployment with three nodes. A periodic ping measurement is configured between every pair of GENI nodes. The count (number of packets to send) value is 1, interval (inter-packet delay) is 1 second, and the time period of the periodic ping experiment is set to 1 minute. We can see that although the memory consumption of the measurement tool is quite low, it increases in a linear fashion. The CPU utilization as shown in Fig. 9 is low, with occasional spikes possibly due to sensor bindings on new rounds of ping invocation.

Fig. 11 and Fig. 12 show the disk and traffic usage. This experiment involved a 5-node GENI deployment with periodic ping measurement requests between all pairs of nodes. A count value of 10 and interval value of 1 second were used in the experiment. Three significant disk and traffic spikes can be seen on the sensing information manager (SIM) corresponding to the data collection phase via scp (secure copy tool) from the sensor pods. However, overall, we find our system footprint to be quite small.

TABLE I
PARAMETER SETTINGS FOR THE TOOLS USED. *(PERIOD: TIME PERIOD BETWEEN INVOCATIONS OF THE MEASUREMENT IN MINUTES)

Tool	Parameters	Values	Description
Ping	Period	10	*
	Interval	1	The time interval in seconds between successive ICMP echo request packets (ping packets)
	Count	10	The number of ping packets sent in each invocation
Traceroute	Period	10	*
PathChirp	Period	10	*
Pathrate	Period	10	*
Spruce	Period	10	*
	Capacity	100 MB	Rough estimate of the path capacity
Tulip	Period	10	*
	Lag	10	Number of measurement rounds used by the tool on each router along the path
	Count	10	The amount of time (in milliseconds) the tool sleeps between successive measurement rounds

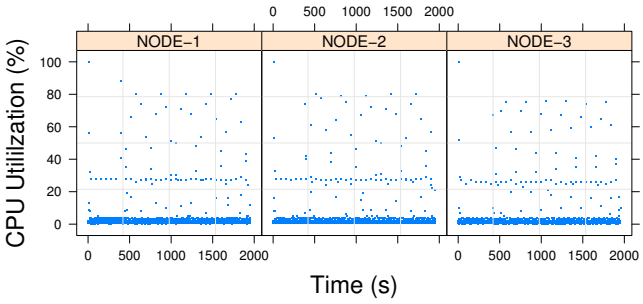

Fig. 9. CPU utilization in a 3-node ping experiment with S^3 Monitor

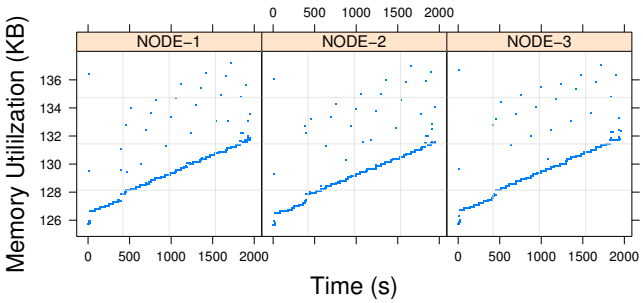

Fig. 10. Memory usage in a 3-node ping experiment with S^3 Monitor

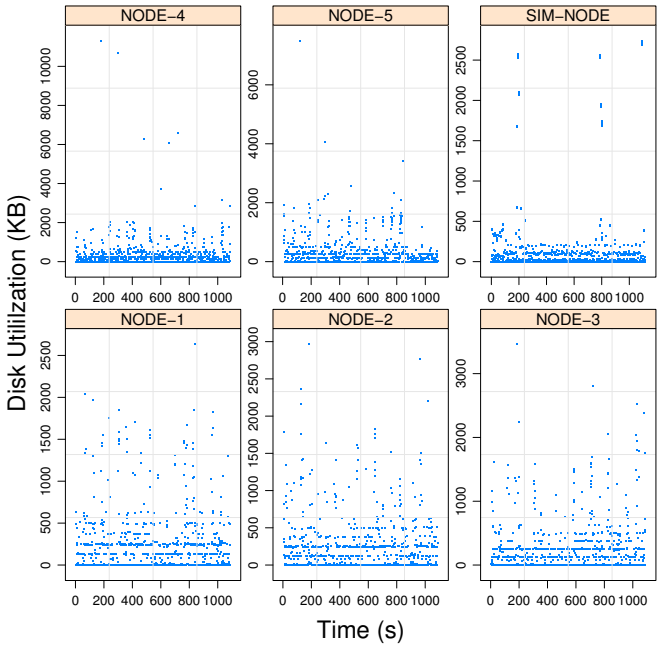

Fig. 11. Disk usage in a 5-node experiment with S^3 Monitor

C. Overhead of Sensor Pod Framework and Sensors

The S^3 Monitor sensor pod includes the framework which communicates with the sensing information manager, and the sensor modules that are dynamically loaded at run-time. To distinguish the performance overhead of these two components, we implement a *zero sensor*. The zero sensor is a mock sensor that has a wrapper to interact with the sensor pod framework but does not consume system resources. We compare the CPU and memory footprints of the zero sensor with the ping and traceroute sensors in Fig. 13 and Fig. 14. The experiment is conducted with the same periodic measurement settings for all three sensors. We note that there is no significant difference in the overhead patterns. Based on these preliminary results, we conclude that that ping and traceroute sensors are lightweight and do not add significant overhead over the overhead of the sensor pod framework. Other sensors are expected to consume more resources.

VI. DEPLOYMENT EXPERIENCE

In this section, we discuss the lessons learned during the design and deployment of the S^3 Monitor service on GENI.

A. Importance of Agile Software Development Practices

The S^3 Monitor service has been developed in parallel with advances and changes in the APIs and support services of the GENI control frameworks. For instance, the RSpec schema has undergone two substantial revisions and multiple minor iterations since the initial implementation of S^3 Monitor

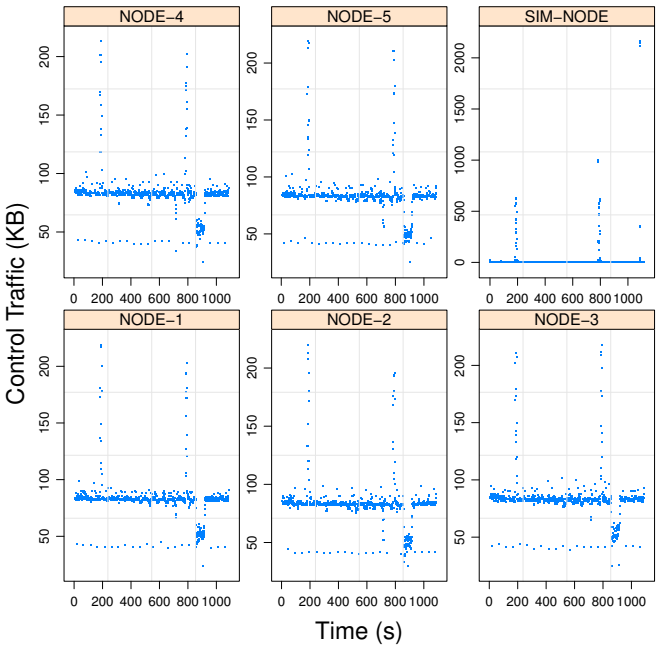

Fig. 12. Control traffic in a 5-node experiment with S^3 Monitor

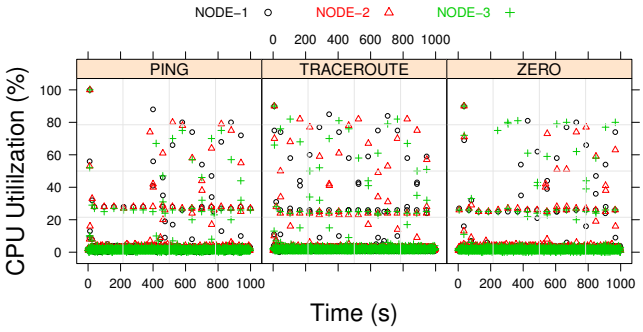

Fig. 13. CPU usage comparison of sensors

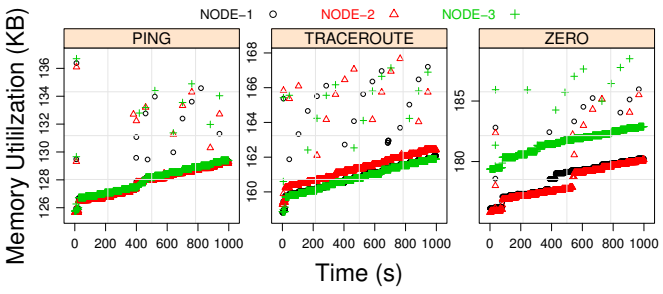

Fig. 14. Memory usage comparison of sensors

components. Similarly, during the course of development, the control frameworks were enhanced to support additional operating system images. Being able to track the moving target of requirements and APIs is very important. Hence, we had to adopt an *agile software engineering approach* [30], [31] for the implementation. This alleviated the overhead and development cost associated with the software development to support the control framework changes. As mentioned earlier, the GENI *Instrumentation and Measurement (I&M)* working group [26] has been working on standardizing the protocols and data formats for exchanging measurement data. The agile development methodology will also aid in making the S^3 Monitor service compliant with the I&M standards as they are released.

B. Balancing Ease of Use and Flexibility

There is a wide variety of experimenters and users of GENI with varying support requirements from GENI infrastructure services. Depending on the goal of the experiments they design, they impose differing requirements and constraints on the GENI infrastructure services, including monitoring services such as S^3 Monitor. While some users require an easy to use graphical interface to configure, control and query S^3 Monitor measurements, others require a more flexible measurement system that allows programmatic (*e.g.,* using scripts) access to control and query measurements. On the one hand, the monitoring system's ease-of-use reduces the deployment and use barriers for the experimenters. On the other hand, flexibility allows the users to craft their monitoring requests to meet the needs of the experiment. The S^3 Monitor service was designed to balance the ease-of-use and flexibility requirements. The APIs to core components of the S^3 Monitor service were developed with maximum flexibility. These flexible APIs were used to develop graphical interfaces for most common usage patterns of the users. In the future, we plan to support pre-formed templates/macros to further simplify common operations. We also designed the APIs of the core services visible to the users keeping in mind advanced monitoring flexibility.

C. Semantic Error Reporting

The GENI experimenters use network measurement services such as S^3 Monitor for two primary purposes: (1) to study the performance impact of various network architectures and conditions in their experiment, and (2) to detect/diagnose the network path metrics during the course of their experiments. Many experiments entail injecting network faults and loss events. Hence, the S^3 Monitor service should be able to correctly diagnose measurement issues that are due to injected errors and problems with the infrastructure. The results from this diagnosis need to be appropriately reported back the user.

Similarly, measurement failures encountered by the users can be due to different reasons. Failures can be the result of failure of the sensing information manager to access sensor pods resident on end-nodes, or due to a problem with the measurement sensor itself. Hence, the S^3 Monitor service's

error reporting feature should correctly report the source of the problem. It is particularly important for the extensible sensor pod, which may have user-provided sensors, to distinguish the two cases.

The S^3 Monitor service uses a web services-based interface to manage measurement tasks on the sensor pods. Several measurement tools (such as network path capacity estimation tools) can take several seconds to finish under normal circumstances. Therefore, *http fetch* timeouts should be set appropriately, so as not to mistakenly confuse the long delay waiting for tool completion with a failure to access the sensor pod.

The sensor pod implementation deployed on GENI is architected such that it can report error causes to the user. For example, the sensor wrappers not only filter out metrics reported by the tool, but also capture errors of execution and report these errors to the user. We must also periodically test the connectivity between the sensing information manager and the sensor pods in the slice, and report errors about failed or disconnected sensor pods.

VII. Related Work

The GENI I&M working group [26] has had strong participation from several GENI projects. Below, we describe the three most relevant projects to ours: (1) OnTimeMeasure [32], (2) LAMP [33], and (3) INSTOOLS [34].

Calyam *et al.* developed the OnTimeMeasure service for GENI [32]. The service is based on their earlier work on measurement orchestration [4], [35]. OnTimeMeasure is the closest GENI measurement service to ours. Like our system, OnTimeMeasure supports active measurements, and its basic orchestration service is similar to our sensing information manager. Unlike our service, their current system does not support extensible sensors.

Leveraging and Abstracting Measurements with perfSONAR (LAMP) [33] is a project to create an instrumentation and measurement system, based on perfSONAR, for use by experimenters on GENI. LAMP is spearheading the I&M effort to standardize a format for data storage and exchange. The perfSONAR service on which LAMP is based allows for the collection of a limited set of active measurements, but does not provide a facility for handling specific user requests or scheduling measurements based on user requirements.

Instrumentation Tools for a GENI Prototype (INSTOOLS) [34] is a project to create a GENI-enabled testbed based on the existing University of Kentucky Edulab. INSTOOLS instrumentation capabilities enable GENI users to better understand the run-time behavior of their experiments. Currently, INSTOOLS provides data collection capability for passive measurements (*e.g.,* SNMP). Supported metrics include traffic statistics, network utilization, and host statistics. INSTOOLS does not invoke active measurements.

In addition to GENI instrumentation projects, several measurement infrastructures have been developed and prototyped on other networks or platforms, *e.g.,* PlanetLab [10] or other Internet hosts. We briefly discuss two systems below: ScriptRoute [3] and iPlane [1].

TABLE II
A comparison of selected features of related measurement infrastructures.

Name	Active / Passive	On-Demand	Standard Tools	User Tools
S^3 Monitor	Active	*yes*	*yes*	*yes*
OnTimeMeasure	Active	*yes*	*yes*	*no*
INSTOOLS	Passive	N/A	N/A	N/A
LAMP	Both	*no*	*yes*	*yes*[1]
Scriptroute	Active	*yes*	*no*	*yes*[2]
iPlane	Active	*no*	N/A	N/A

[1] LAMP "measurement points" communicate using a well-defined protocol, and a measurement tool can be added by providing an interface in this protocol.
[2] Scriptroute can execute user-defined tools, provided they are implemented in the Scriptroute language.

Spring *et al.* propose a public measurement facility, *Scriptroute* [3], which allows individual users, network managers, or researchers to conduct network measurements on a shared measurement mesh, controlled by security and rate-limiting filters. Scriptroute accepts arbitrary measurements at arbitrary times and blindly enforces administrative bandwidth and security filters, potentially invalidating measurements, before the probes even leave the measurement host. The measurement tools invoked by Scriptroute must be implemented in a scripting language using Scriptroute-provided primitives for sending and receiving packets, and the system is not capable of invoking third-party measurement tools. Scriptroute lets individual users select, configure, invoke, and use the measurement tools. Our framework eases some of this burden on users, while retaining the flexibility of using custom measurement tools.

iPlane [1] performs scalable continuous measurements over a set of Internet paths, and uses them to perform prediction of path properties. Unlike our system, iPlane performs no measurements on-demand. Hence, it is unsuitable for troubleshooting transient faults in progress or other tasks which require real-time results. It does not necessarily invoke standard measurement tools or present their results to users in a direct fashion.

Table II compares the measurement infrastructures discussed in this section. The type of measurements the infrastructure manages (active or passive) is indicated, as well as whether the infrastructure is capable of invoking measurements on-demand, at user-specified times. The final two columns, Standard Tools and User Tools, indicate whether the measurement infrastructure is capable of invoking standard third-party measurement tools (versus requiring specially implemented tools) and whether the infrastructure is capable of invoking user-supplied tools, respectively.

VIII. Conclusions and Future Work

In this paper, we have presented the requirements, design, and performance of the S^3 Monitor measurement service deployed on the Global Environment for Network Innovations (GENI). The service is targeted at GENI experimenters and operators, and therefore must satisfy their requirements. Through

the emphasis on portability and extensibility, we believe that our service is quite versatile and caters to a wide variety of user needs. Our evaluation of the system shows that the response time of the service is rapid (unless the measurement tool itself is time-consuming), and the system footprint is small.

Our future work plans include a focus on safety. By safety, we mean that users cannot trigger arbitrarily large measurement probes that exceed a specified budget. The careful admission control of measurement requests can allow the administrators to prioritize the measurements they admit according to their policies [36]. We also plan to incorporate our work on measurement inference for increased scalability [37], [38], [39] and our work on scheduling to reduce interference among active measurements [40]. We are also currently integrating our S^3 Monitor system with the Instrumentation Tools (INSTOOLS) [34] service on GENI.

ACKNOWLEDGMENTS

This work has been supported in part by GENI project 1723. The authors would like to thank Nagarjuna Reddy (HP), Prashanthi Gaddam (HP), and Ranjan Kumar (HP) for their help with the software development, and Nabeel Butt (Purdue University) for his help with the experimental evaluation. Thanks also to the HP Labs S^3 team for providing access to the S^3 codebase, and to Vic Thomas and Luisa Nevers of the GENI Project Office for their valuable feedback.

REFERENCES

[1] H. V. Madhyastha, T. Isdal, M. Piatek, C. Dixon, T. Anderson, A. Krishnamurthy, and A. Venkataramani, "iPlane: An information plane for distributed services," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, Nov. 2006, pp. 367–380.

[2] A. Nakao, L. Peterson, and A. Bavier, "A routing underlay for overlay networks," in *Proceedings of ACM SIGCOMM*, 2003, pp. 11–18.

[3] N. Spring, D. Wetherall, and T. Anderson, "Scriptroute: A public internet measurement facility," in *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS)*, 2002.

[4] P. Calyam, C. Lee, E. Ekici, M. Haffner, and N. Howes, "Orchestration of network-wide active measurements for supporting distributed computing applications," *IEEE Transactions on Computers (TOC)*, vol. 56, no. 12, Dec 2007.

[5] H. Song, L. Qiu, and Y. Zhang, "NetQuest: A flexible framework for large-scale network measurement," in *Proceedings of ACM SIGMetrics/Performance*, Jun. 2006.

[6] H. Song and P. Yalagandula, "Real-time end-to-end network monitoring in large distributed systems," in *Proceedings of IEEE COMSWARE*, 2007.

[7] N. Hu and P. Steenkiste, "Exploiting Internet route sharing for large scale available bandwidth estimation," in *Proceedings of ACM IMC*, 2005.

[8] "The Global Environment for Network Innovations (GENI)," http://geni.net/.

[9] P. Yalagandula, P. Sharma, S. Banerjee, S. Basu, and S.-J. Lee, "s^3: A scalable sensing service for monitoring large networked systems," in *Proceedings of the ACM SIGCOMM Workshop on Internet Network Management (INM)*, September 2006.

[10] Planetlab, "PlanetLab Home Page," http://www.planet-lab.org/, 2011.

[11] V. J. Ribiero, R. H. Reidi, R. G. Baraniuk, J. Navratil, and L. Cottrell, "pathChirp: Efficient available bandwidth estimation for network paths," in *Proceedings of the Passive and Active Measurement (PAM) Workshop*, 2003.

[12] M. Jain and C. Dovrolis, "Pathload: A measurement tool for end-to-end available bandwidth," in *Proceedings of the Passive and Active Measurement (PAM) Workshop*, Mar. 2002.

[13] J. Strauss, D. Katabi, and F. Kaashoek, "A measurement study of available bandwidth estimation tools," in *Proceedings of the ACM IMC 2003*, Miami, FL, October 2003.

[14] C. Dovrolis and P. Ramanathan, "Packet dispersion techniques and capacity estimation," *IEEE/ACM Transactions on Networking*, December 2004.

[15] R. Kapoor, L. Chen, L. Lao, M. Gerla, and M. Y. Sanadidi, "CapProbe: A simple and accurate capacity estimation technique," in *Proceedings of SIGCOMM*, Aug. 2004.

[16] R. Carter and M. Crovella, "Measuring bottleneck link speed in packet-switched networks," Boston University, Tech. Rep. BU-CS-96-006, Mar. 1996.

[17] S. Keshav, "A control-theoretic approach to flow control," in *Proceedings of SIGCOMM*, 1991.

[18] R. Jain and S. Routhier, "Packet trains: Measurements and a new model for computer network traffic," Defense Technical Information Center, Tech. Rep., Nov. 1985.

[19] "Scalable Sensing Service Monitor," http://groups.geni.net/geni/wiki/ScalableMonitoring.

[20] "The ProtoGENI Control Framework for GENI Cluster C," http://www.protogeni.net/.

[21] "A Prototype of a Million Node GENI," http://groups.geni.net/geni/wiki/MillionNodeGENI.

[22] "The Global Environment for Network Innovations (GENI) Resource Specification (RSpec)," http://groups.geni.net/geni/wiki/GeniRspec.

[23] R. Mahajan, N. Spring, D. Wetherall, and T. Anderson, "User-level Internet path diagnosis," in *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2003.

[24] G. van Rossum, "Python reference manual," Centrum voor Wiskunde en Informatica (CWI), Tech. Rep. CS-R9525, May 1995.

[25] "Twisted.web event-driven web service framework," http://twistedmatrix.com/.

[26] "The Global Environment for Network Innovations (GENI) Intrumentation and Measurement Work in Progress (I&M)," http://groups.geni.net/geni/wiki/GeniInstMeas.

[27] "Collectl," http://collectl.sourceforge.net/index.html/.

[28] "Collectl performance," http://collectl.sourceforge.net/Performance.html/.

[29] "Selenium WebDriver," http://seleniumhq.org/projects/webdriver/.

[30] K. Beck, M. Beedle, A. van Bennekum, A. Cockburn, W. Cunningham, M. Fowler, J. Grenning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas, "Manifesto for agile software development," http://agilemanifesto.org/, 2001.

[31] K. Beck, *Implementation Patterns*, 1st ed. Addison-Wesley Professional, 2007.

[32] "OnTimeMeasure: Centralized and distributed measurement orchestration software," http://groups.geni.net/geni/wiki/OnTimeMeasure.

[33] "Leveraging and abstracting measurements with perfSONAR (LAMP)," http://groups.geni.net/geni/wiki/LAMP.

[34] "Instrumentation tools for a GENI prototype," http://groups.geni.net/geni/wiki/InstrumentationTools.

[35] P. Calyam, C. Lee, P. K. Arava, and D. Krymskiy, "Enhanced EDF scheduling algorithms for orchestrating network-wide active measurements," in *Proceedings of the 2005 IEEE Real-Time Systems Symposium (RTSS)*, 2005.

[36] E. Blanton, S. Fahmy, and S. Banerjee, "Resource management in an active measurement service," in *Proceedings of the IEEE Global Internet Symposium*, Apr. 2008.

[37] S. Gangam and S. Fahmy, "Distributed partial inference under churn," in *Proc. of IEEE Global Internet*, 2010.

[38] E. Blanton, S. Fahmy, and G. N. Frederickson, "On the utility of inference mechanisms," in *Proc. of IEEE ICDCS*, 2009, pp. 256–263.

[39] E. Blanton, S. Fahmy, G. N. Frederickson, and S. Gangam, "On the cost of network inference mechanisms," *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, vol. 22, no. 4, pp. 662–672, April 2011.

[40] S. Gangam and S. Fahmy, "Mitigating interference in a network measurement service," in *Proc. of IEEE IWQoS*, 2011.