A Passive Measurement System for Network Testbeds

Charles Thomas, Joel Sommers, Paul Barford, Dongchan Kim, Ananya Das, Roberto Segebre, and Mark Crovella

University of Wisconsin, Colgate University and Boston University
{cthomas,dkim24}@wisc.edu, {jsommers,adas,rsegebre}@colgate.edu,
{crovella,pb}@cs.bu.edu

Abstract. The ability to capture and process packet-level data is of intrinsic importance in network testbeds that offer broad experimental capabilities to researchers. In this paper we describe the design and implementation of a passive measurement system for network testbeds called GIMS. The system enables users to specify and centrally manage packet capture on a set of dedicated measurement nodes deployed on links in a distributed testbed. The first component of GIMS is a scalable experiment management system that coordinates multi-tenant access to measurement nodes through a web-based user interface. The second component of GIMS is a node management system that enables *(i)* local processing on packets (*e.g.,* flow aggregation and sampling), *(ii)* meta-data to be added to captured packets (*e.g.,* timestamps), *(iii)* packet anonymization per local security policy, and *(iv)* flexible data storage including transfer to remote archives. We demonstrate the capabilities of GIMS through a set of micro-benchmarks that specifically highlight the performance of the node management system deployed on a commodity workstation. Our implementations are openly available to the community and our development efforts are on-going.

1 Introduction

Network testbeds are designed to offer environments to researchers and practitioners in which experimental systems, configurations and protocols can be carefully tested and evaluated. Network testbeds in use today can be differentiated by the specific systems, level of control and "realism" that they offer users. While the strengths and weaknesses of different testbed types have been well documented, the utility of each depends directly on the ability to gather *measurements* from the infrastructure.

Measurements in network testbeds can be broken into two categories: active and passive. Active measurements are based on sending and receiving specifically crafted packet probes through the infrastructure (*e.g.,* `traceroute` measurements). These probes enable a variety of characteristics to be measured such as end-to-end delay, loss and jitter [22]. While active probe-based measurements are important and widely used in testbeds and operational networks, they may

T. Korakis, M. Zink, and M. Ott (Eds.): TridentCom 2012, LNICST 44, pp. 130–145, 2012.

lack detail or precision or be entirely unable to capture aspects of behavior that are critical for experiments.

Passive measurements are based on using specialized counting or capture mechanisms that are built into software and systems deployed in network testbeds. Standard examples are log files from servers, flow-export logs [17, 26] or the diverse measurement information bases (MIBs) that are available from networked devices via the Simple Network Management Protocol (SNMP) [15]. One of the most compelling types of passive measurement is the ability to capture packet information from transmissions on links in a testbed. Information from packet traces can be critical to experiments with new network applications, protocols and security techniques, as well as for day-to-day management and troubleshooting of the testbed infrastructure itself.

There are a myriad of challenges to enabling packet capture capability within a network testbed. First, packet capture almost always requires dedicated systems since measurements on high-bandwidth links can result in overheads that are beyond the capability of standard hardware. This means that sufficiently capable systems must be acquired, configured, deployed and (securely) managed alongside the experimental systems. Further, if packet capture is meant to be available to concurrently running experiments, the measurement systems must be able to log data such that multiple tenants have exclusive access to their own data. Finally, packet capture always has security and privacy implications since packets can contain personally identifiable and private information. Depending on the size and diversity of the testbed, these challenges can become quite significant.

In this paper we describe a packet capture management environment for network testbeds called GIMS (GENI Instrumentation and Measurement Systems). The system was designed for deployment within the GENI infrastructure [4], however it has evolved into a system that can be independent from that environment. GIMS provides the capability to *(i)* configure packet capture measurements (typically associated with an experiment on the testbed) on a deployed set of dedicated packet capture devices, *(ii)* manage measurements from simultaneous users, *(iii)* enforce local security and privacy policies, and *(iv)* summarize and archive captured data to remote storage devices.

The architecture of GIMS is divided into three major components. The first is the GIMS front end, which includes web-based user interfaces for GIMS administrators and users, and allows the user access to monitoring capabilities during the experiment and access to results after an experiment is over. The second component of the architecture is the GIMS backend, which instantiates and facilitates control of measurements, and coordinates activities between the front end (*i.e.,* admins and users) and the GIMS packet capture devices. The backend also includes a monitoring system that gathers and stores system information and logs from running experiments. The third component is the packet capture control system that runs on the packet capture devices that are deployed within the testbed. This control system includes the capability to enforce privacy policies, summarize and aggregate packet data, and archive data in a variety of

ways for multiple simultaneous experiments. Communication throughout GIMS is facilitated through an XML/RPC-based command language. The system is designed for scalable and extensible deployment and to simplify tasks of deploying capture devices, and managing and using the system. The GIMS components were implemented in run on commodity PC hardware and UNIX-based operating systems.

We describe the implementation of GIMS and include a screenshot of the front-end user interface, and show results from a set of micro-benchmarks on the packet capture control system. The former highlights some the capabilities of the system from the user perspective. The latter is critical for understanding the behavior and capabilities of the components that will actually be capturing experiment data in the testbed. Our microbenchmark results show that the packet capture system performs well under high offered packet rates, with zero or nearly zero packet loss under a load of 200K packets/sec. In a configuration with realistic TCP traffic offered at an average rate of 500 Mb/s and in which the capture system produced flow export records, we observed zero packet loss.

The GIMS software distribution is openly available to the community [9]. It has been running in a prototype deployment within the Wisconsin Advanced Internet Lab [8] for nearly a year. We are currently in the process of identifying locations for deployment of packet capture nodes in different parts of the GENI infrastructure.

The remainder of this paper is organized as follows. In Section 2, we provide details on the GIMS architecture. In Section 3, we provide details on the implementation of GIMS. We evaluate the packet capture system through a set of micro-benchmarks in Section 4. Studies and projects related to GIMS are described in Section 5. We summarize, conclude and describe our future work in Section 6.

2 System Requirements and Architecture

In this section, we describe the requirements and design of GIMS. At the highest level, development of GIMS was motivated by the recognition that the ability to capture packets on links (*i.e.,* passive measurement) in a network enables a broad range of network research experiments and network operations activities. These include experiments with new networking protocols, experiments in network security, experiments with new network applications, experiments with new measurement tools, etc. From a network operations perspective, passive packet capture enables network performance tuning and network troubleshooting. However, this capability is not common in network testbeds and coordinated management of packet capture devices is complex.

2.1 GIMS Relationship to GENI

GIMS was developed as a measurement infrastructure that would be deployed within GENI [4]. To that end, the primary requirements beyond specific experimental functionality is that it be consistent with the GENI authorization and

credentialing mechanisms and that it interface with one of the GENI control frameworks.

GIMS currently implements interfaces for the ProtoGENI control framework as discussed in Section 3. However, during the development process, every effort was made to make the implementation general and to modularize components such that interfaces to additional control frameworks could be easily developed, and so that the system could be used in a standalone fashion. A standalone version of the system is nearing completion.

2.2 Passive Measurement System Requirements

The requirements for the GIMS environment are based on the GENI Instrumentation and Measurement Systems Specification [18]. That document specifies a broad vision for instrumentation and measurement, and discusses the trade-offs and challenges for different types of instrumentation and measurement within GENI. Our focus is specifically on the objective of developing a *passive packet capture* capability *i.e.,* the ability to gather, save and analyze packets from taps on links in a network testbed.

The general requirements of the GIMS environment include (a number borrowed from [18]):

- Full or partial packet capture at line rate with zero packet loss,
- No (or at least measurable) impact on experiments,
- Extensibility (*i.e.,* the ability to add new measurement synthesis capability),
- High availability (*e.g.,* at least as available as testbed systems on which experiments are conducted),
- Large capacity (*i.e.,* the ability to support a diverse set of simultaneous activities from a large number of experiments),
- Remote management and monitoring capability,
- Access control (*i.e.,* the ability to specify what data is available from a particular device or collection of devices, to whom, and for how long),
- Flexible storage including the ability to house data locally and to stream to a remote archive,
- Measurement nodes that are secure from unauthorized external access,
- Ease of use for both administrators and users of the system,
- Deployment on commodity PC's and UNIX-based operating systems.

Specific requirements for the packet capture systems include:

- Support for IPv4 (IPv6 support is future work),
- Support for IPv4 header capture only,
- Support for specifying individual fields of interest in the IPv4 header,
- Support for on-the-fly prefix preserving anonymization of specified fields,
- Support for no-loss packet capture at line rate of at least 1 Gbps,
- Support for up to 256 simultaneous active experiments per node.

Satisfying these requirements entails the development of a packet capture management environment that bears many similarities to network testbeds themselves and can easily be thought of as a parallel experimental infrastructure. We are aware of no other packet capture management systems that fully satisfy these requirements.

2.3 Architectural Specification

The design space for GIMS makes several assumptions. First, dedicated measurement hardware will be deployed in a network testbed as illustrated in Figure 1.

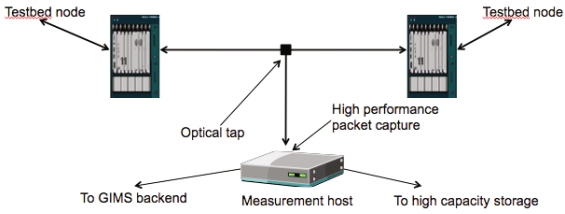

Fig. 1. The basic physical components that support a GIMS deployment

These measurement host systems are connected to target links via taps (optical splitters or active devices) or SPAN ports on switches. The systems are commodity PC hosts running a UNIX-based OS and may include high performance packet capture cards (*e.g.,* such as Endace [3]), or specialized software to improve packet capture efficiency (*e.g.*, [14, 16]). These systems are remotely accessible via a management network interface and include a separate network interface for streaming captured data to remote, high capacity storage systems. While not an explicit part of the GIMS requirements, attaching high precision timestamps to packets is highly desirable for different types of experiments. This is most effectively accomplished with specialized hardware and GPS support, although there are emerging software solutions [28]. Adding high precision timestamps is a future objective for GIMS deployments but does not require any specific capability from our software.

The GIMS design is three-tiered and divided into components as shown in Figure 2. The first component is the *front end*. This component of the system includes the interface mechanisms for both GIMS users and administrators that enable access to the measurement infrastructure. The interfaces enable users to configure their packet capture measurements. This includes specifying the set of nodes that will be monitored [1], IP aggregates that will be captured, remote data

[1] Users currently must have out-of-band knowledge of the links their experimental traffic will use. Coordinating links that have packet capture nodes with testbed experiments requires specific coordination with the testbed management infrastructure.

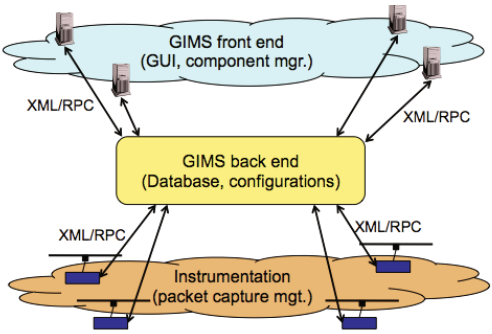

Fig. 2. The key components in the GIMS architecture

storage targets (*e.g.,* Amazon S3) and local packet processing (*e.g.,* sampling or flow aggregation). The admin GUI enables access to and configuration of remote measurement nodes. The front end also includes the *component manager*. This system enables coordination and integration of GIMS with a network testbed control system such as ProtoGENI. The component manager facilitates authenticated access to GIMS and control of GIMS systems from the testbed system should the GIMS GUI's not be required. In short, it makes GIMS more broadly useable across diverse testbed infrastructures. The front end can run on a single system or on multiple systems in a distributed testbed.

The second component in the GIMS design is the *back end*. This component of the system is responsible for coordination of front end systems and the distributed measurement nodes; it is, in a sense, the "nerve center" of GIMS. The back end provides information about the availability and status of measurement nodes via GIMS control messages to the front end. It also facilitates all communication via GIMS control messages with measurement nodes, including registration with the infrastructure, administrative configuration, and configuration of packet capture activities for individual users. The core component of the back end is a database system that maintains all state information about the configurations of GIMS systems. This component can be distributed to enhance robustness and to enables GIMS to scale to support a large infrastructure.

The third component in GIMS design is the *packet capture management system* that runs on each of the measurement nodes that are deployed on links in the testbed. Fundamentally, these nodes run some kind of packet filter (*e.g.,* `libpcap`) that enables packets to be captured from a network interface, locally processed, and stored. The packet capture management system runs on these nodes and *(i)* facilitates multi-tenant use of measurement systems, *(ii)* enforces local privacy policies, *(iii)* provides data summarization and aggregation capability, and *(iv)* provides data streaming to a designated remote storage device. Remote storage recognizes the fact that packet capture on high speed links can quickly fill disks, thus each user is only allocated a fixed local storage volume. It is also critical that the packet capture management system impose a very low

processing overhead on the measurement nodes so as not to affect packet capture and processing and thereby impact measurements.

The control and data flow in GIMS is illustrated in Figure 3. Users specify experiments and their associated measurement configurations through the GIMS GUI or an interface in the testbed control framework. Among other things, this results in filters deployed on measurement systems and storage allocation. As experiments are run in the testbed, packets are captured and stored in the user-specified archive. At the conclusion of the experiment, the filters and storage are deallocated. The front end and back end facilitate the flow of control data and maintain the current state of the measurement systems.

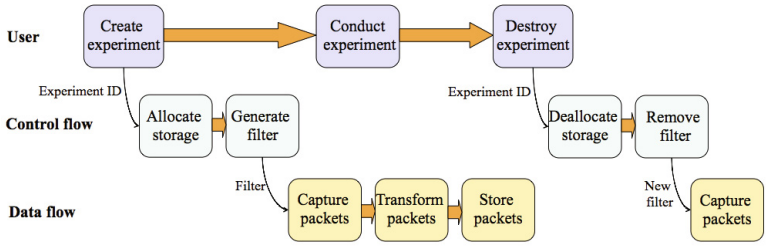

Fig. 3. Control and data flow in GIMS

Finally, the security and access control are critically important in GIMS since measurement nodes may be deployed on live links. Measurement nodes are therefore only accessible to authorized users or administrators. Data collection on the measurement nodes enforces local privacy policies by only capturing designated fields of packets (*e.g.,* header fields) and applying any required anonymization before transmitting to either local or remote storage. The front end, back end and measurement systems themselves are secured from unauthorized external access by only allowing access on specified ports from specified systems and from users with the correct credentials.

3 Implementation Details

In this section we describe implementation aspects of GIMS. We first discuss the management and control components of GIMS (the front-end and back-end components), followed by the packet capture subsystem implementation.

Figure 4 depicts the GIMS system components and interfaces among them. These components are consistent with the architectural specification described in Section 2. The user-facing components offer web-based views for users to manage, configure, and control testbed measurements. These front-end components communicate with the back-end, which coordinates all of the activities in the GIMS infrastructure. As such, the back-end performs remote procedure calls to

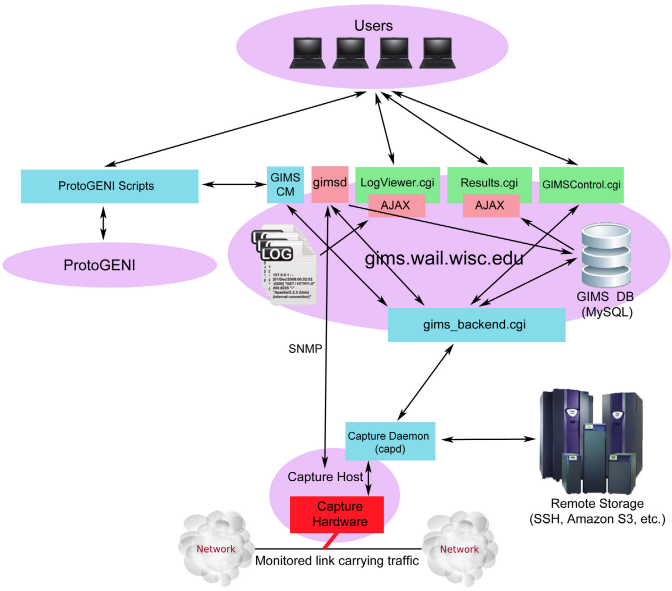

Fig. 4. The GIMS system components

configure and control capture daemon components, and to manage the database of users and device configurations in response to user and administrator actions initiated from the front-end.

3.1 GIMS Front End

The most prominent of the front-end components, the GIMS control GUI, is implemented using AJAX and runs from a web server in the testbed (in this case `gims.wail.wisc.edu`). The GUI allows users to create device configurations, assign names to specific configurations, and store them in the GIMS database for retrieval. A screenshot of the control GUI is shown in Figure 5. A new set of device configurations can be created automatically when an experiment in the network testbed is created (*e.g.,* a GENI slice), or it can be manually generated. Each configuration can be saved in the GIMS database for future retrieval and modification.

To obtain the status of a set of measurements, or an ongoing measurement session, a GIMS user can view a separate results page, which can either show the full results of a measurement session after it has ended, or display some (near-)real-time information. The view of an ongoing measurement allows a user to see a status log, as well as the results of querying the capture daemons for basic statistics of packet or flow capture.

A separate administration GUI allows authorized users of GIMS to add, delete and edit devices in the GIMS back-end database. A device must be added to the

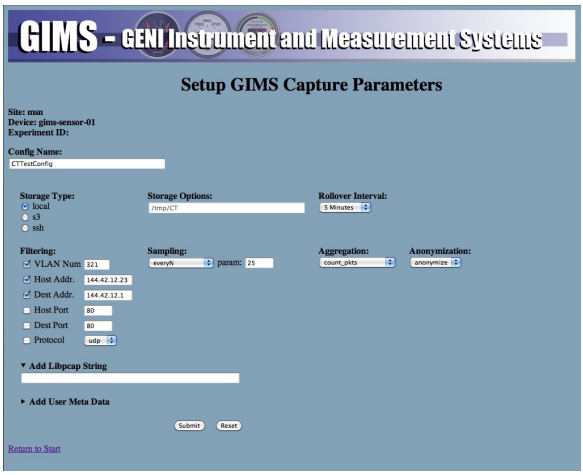

Fig. 5. The GIMS control GUI

database before it will be available to be selected for an experiment. The user is prompted for information about the device such as location, device name, device type, hostname, port and description. The capabilities of the device also need to be entered so that the correct options can be displayed to the user during the creation of device configurations.

3.2 GIMS Back-End

The GIMS back-end implements various functions to coordinate user actions with the capture daemon systems deployed in a testbed. There are also a number of functions to support generation, update, and retrieval of users' device configurations in the GIMS database, and it performs a variety of sanity-checking, error-checking and logging functions.

The GIMS database is a core back-end component and is built on MySQL. It contains tables to keep track of experiment state, location and type of devices, which devices are being used in a given experiment, the configuration of experiment devices, and statistics for each experiment.

Lastly, the "gimsd" monitoring daemon uses SNMP to monitor GIMS capture devices that are currently running and collecting data on behalf of users. It takes a snapshot of system performance every 15 seconds and stores the results in the GIMS database. These results are available to the user via the GIMS Results tool during and after experiment execution.

3.3 GENI Integration

GIMS currently includes a collection of scripts that forms a modular interface to facilitate interaction with the ProtoGENI control framework. These interfaces

enable integration with the GENI testbed, and their modular construction enables additional control frameworks to be added without affecting other GIMS components. These scripts are used to control and query various aspects of the ProtoGENI system from GIMS. Likewise, the GIMS component manager is a GIMS front-end component that translates ProtoGENI control instructions into GIMS-specific control actions. In general, these components deal with translation of GENI contructs of "slices" and "slivers" into GIMS components and configurations. Because of the modular design of this part of the back-end, it is fairly straightforward to integrate with different external control frameworks. As noted above, GIMS can also operate autonomously (*i.e.,* without an interface to a specific control framework).

3.4 Packet Capture Subsystem

The core measurement capabilities of GIMS are implemented in a subsystem called the *capture daemon*. This subsystem consists of three components: the capture daemon controller, which handles requests from the GIMS backend to configure and control packet capture processes, the storage controller, which handles storage interactions, and the capture daemon itself, which performs the packet capture, aggregation and transformation of packets, and creation of metadata. The capture daemon subsystem is depicted in Figure 6.

The capture daemon controller implements a set of XML/RPC handlers to accept requests from the GIMS back end for configuring packet capture system parameters and storage parameters, for starting, stopping, and pausing capture daemon processes, and for gathering some statistics on the progress of packet

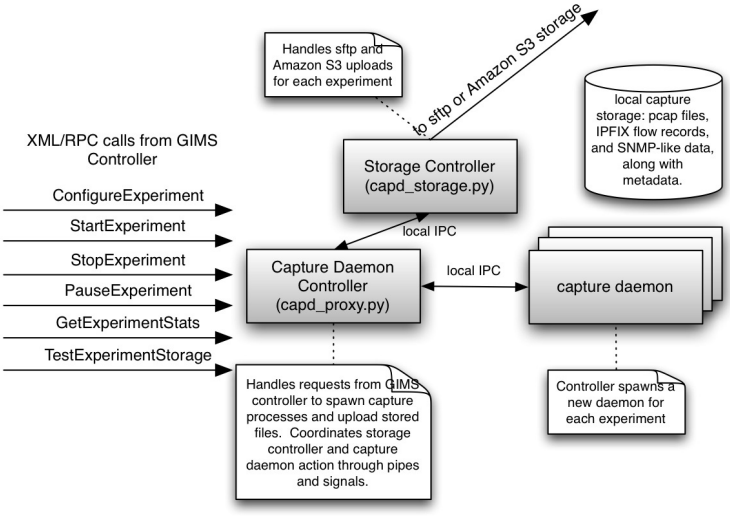

Fig. 6. GIMS capture daemon architecture

capture. It is implemented in Python as a single process, and communicates with the storage controller and capture daemon processes through standard UNIX inter-process communication mechanisms (pipes and signals). It also performs detailed logging, which the GIMS backend relays to users if they want to gain detailed runtime information about the capture system.

The storage controller is also implemented in Python as a single process, and utilizes various modules to interact with `sftp` and Amazon S3 servers. As new files are produced and become available for transfer, the storage controller initiates upload to user-specified directories, in the case of sftp, or a user-specific bucket, in the case of S3. Since there may be multiple capture processes running simultaneously we must take care to avoid blocking operations in the storage daemon. Thus, the files generated from each configuration are handled by separate threads in the storage controller.

Lastly, the capture daemon process implements the actual passive measurement collection and processing. It is implemented in C, and uses the standard `libpcap` API for packet capture. As a result, it can leverage any modified versions of `libpcap` that are available for accelerating packet capture, *e.g.*, [3,16]. To perform aggregation into flow records, the capture daemon uses the open source `libfixbuf` and `yaf` libraries, and for anonymization, we use the well-known prefix-preserving IP address anonymization algorithm of Fan *et al.* [19].

As data files are created (either packet traces or flow record traces), the capture daemon generates metadata in an easily processed XML format. The metadata include information about the experiment configuration (obtained through the initial configuration call from the GIMS back-end) as well as user-specific metadata (specified in the GIMS front-end GUI). Finally, experiment-specific details related to the runtime performance of the experiment are added periodically as an experiment proceeds. For example, the number of received packets and bytes are added periodically to the metadata, as well as information about any interface packet drops that `libpcap` reports. These packet and byte counters are also relayed to the GIMS back end, so that a user can gain (near-)real-time information about his or her experiment as it runs.

4 System Microbenchmarks

Understanding the performance of the packet capture system is important to ensure that it does not skew measurements *e.g.,* by inadvertently dropping packets. In this section we describe a set of micro-benchmark experiments to evaluate the performance of the GIMS capture daemon system. We also discuss operational aspects of other parts of the GIMS system.

4.1 Experiments

The goal of our microbenchmark experiments was to evaluate the impact of different configuration parameters on the performance of the GIMS capture daemon. Since the capture daemon can perform optional transformation and

aggregation of packets, we sought to understand the implications of different options. Our testbed consisted of four identical commodity workstations, each with Intel Xeon E5530 quad-core processors, 8 GB of RAM, and dual-port Intel Gigabit Ethernet card dedicated to experiment traffic. A fifth workstation had similar CPU and memory specs, but two dual-port Intel Gigabit Ethernet cards. The four identical hosts were wired directly to the fifth host with cross-over cables, creating a star topology. Each host ran Linux 2.6.32.

On the fifth (center of the star) host, we installed and ran the GIMS capture daemon software. We additionally installed the `PF_RING` [16] kernel module, as well as the modified Intel Gigabit Ethernet (`igb`) driver to improve raw packet capture performance. While experiments were running, we monitored CPU load and collected information (from the capture daemon, and also directly from `PF_RING`) about packet drops. We used three of the other hosts to generate traffic, and the fourth as a traffic sink. The packet capture daemon was configured to monitor traffic on the interface connected to the traffic sink node.

We used `iperf` to generate uniform streams of UDP packets [1], from 64 bytes through 1000 bytes, and the Harpoon traffic generator to generate more realistic TCP traffic [27]. Due to limitations of using `iperf` for generating traffic, we were only able to generate up to 210,000 packets per second (210 Kpps). This rate is approximately the maximum rate of 512 byte packets that can be generated at 1 Gb/s. We configured Harpoon to generate an average load of about 500 Mb/s.

The capture daemon was run in four configurations. In the first, we captured the first 64 bytes of packets and did not perform any other transformations to the packets. In the second, we configured the capture daemon to do uniform sampling of 10% on received packets (*i.e.*, 90% of received packets were discarded). Again, we captured and stored the first 64 bytes of the packets that remained after sampling. In the third configuration, we collected simple SNMP-like packet and byte counters on the received traffic, and in the fourth, we collected and stored flow-level information in the IPFIX format [26]. In each configuration, data was stored on the local system and not streamed to remote storage. We used the `iperf` traffic generator for the first three configurations, and Harpoon in the flow-oriented configuration.

4.2 Results

We first note that in all experiments, CPU load on the capture daemon host was quite low (around 10%). This result is consistent with other measurements that have been performed with `PF_RING`-enabled systems [11, 14], and suggests that the capture daemon does not impose significant processing overhead.

Table 1 shows the percent of packets captured (*i.e.*, not dropped) for the three experiments involving the `iperf` traffic generator. Results are shown for each of the three capture daemon configurations, and five different `iperf`configurations. As noted above, due to limitations of `iperf` we could only generate, at maximum, about 210 Kpps. We see from the table that there was zero or close to zero packet drops in all cases except for experiments in which we collected 64 byte packet headers with 64 byte and 128 byte packet traffic. In our Harpoon experiment

with collecting flow records at the capture daemon, reported packet drops were also nearly zero (less than 0.5%).

In both our CPU utilization measurements and packet drop measurements, our results are consistent with prior work that has studied the performance of `PF_RING`-based systems. As a result, we expect the capture daemon to exhibit scaling properties similar to other systems that use `PF_RING`, and perform well under high-load situations. We also note that the capture daemon can also take advantage of hardware-accelerated packet capture platforms such as the Endace DAG cards [3] to scale to higher speeds, since these platforms often offer modified versions of the `libpcap` software.

Table 1. GIMS capture daemon system performance in different configurations using the `iperf` traffic generator. Table values show percent of packets captured.

Packet size	64	128	256	512	1024
Offered packet rate	200 Kpps	201 Kpps	201 Kpps	208 Kpps	100 Kpps
Packet header capture	96.9	98.9	100	100	100
Header capture; 10% sampling	100	100	100	100	100
SNMP-like counters	100	100	100	100	100

5 Related Work

Passive network measurements are instrumental to operators and researchers for gaining insight into network behavior and performance. As such, there have been many protocols and systems developed over the years to facilitate passive collection of network measurements.

The most ubiquitously deployed measurement capability today is defined by the Simple Network Management Protocol (SNMP) [15]. The SNMP standards define a set of counters and configuration variables arranged in a hierarchical structure, called the Management Information Base, as well as protocols to retrieve counters and set device configurations. For example, many network operators utilize basic byte and packet interface counters that are available through SNMP to gain a coarse view of traffic characteristics in their networks. Operators and scientists also use software packages such as the Multi-router Traffic Grapher (MRTG) [24] to visualize these basic SNMP counters in appealing and useful graphics, and to track traffic patterns over time.

Although SNMP counters are widely available, they cannot provide insight into application traffic patterns or other traffic details. As a result, many routers have the ability to export *flow records*, which contain more detailed information on individual application flows, and there have been other efforts within the IETF to define flexible traffic flow measurement mechanisms, *e.g.*, RTFM [12]. For example, the *de facto* standard Cisco Netflow formats [2, 17] are supported on many devices, and it is likely that the flexible flow record formats provided by the recent IPFIX standard [26] will see broad adoption.

Packet traces provide one of the most detailed views of network traffic, at the cost of higher CPU and I/O load to perform the data collection. There are

many standard tools and libraries available to facilitate capture and processing of packet data, *e.g.*, [7, 13, 23], and specialized hardware platforms that enable efficient packet capture at multi-gigabit speeds [3].

A number of efforts within the research community have focused on improving packet capture efficiency on commodity hardware systems. For example, the `PF_RING` system can be compiled and installed in a Linux kernel to enable packet capture at very high rates [16], and the related `vPF_RING` system facilitates high-speed capture in Linux-based virtual machine environments [14]. There are `PF_RING`-based modifications available to the standard `libpcap` API, enabling applications to transparently take advantage of the performance improvements provided by `PF_RING`. Since the GIMS packet capture daemon is built on top of the `libpcap` interface, it can also transparently take advantage of these performance improvements. Still, tuning a commodity operating system and hardware platform to achieve best performance can be difficult. Braun *et al.* evaluated packet capture bottlenecks on FreeBSD and using `PF_RING` and `PF_PACKET` on Linux in order to provide guidelines for achieving optimal performance [11].

There have been a number of systems developed to improve passive packet measurement capabilities by making it easier for network managers and/or researchers to initiate, collect, process and/or analyze the collected data. For example, the `pktd` system provides authorized users with an API to initiate packet capture on an end host while not having to give the user privileges to directly open a network interface card in promiscuous mode [20]. Building on the capabilities of `pktd`, Agarwal, *et al.* developed a system to remotely initiate packet capture at specific vantage points within an enterprise network for the purpose of debugging performance problems. In a similar vein, Hussain, *et al.* describe a system for passive packet capture based on using Endace DAG cards that also provides automated storage for packets [21]. These systems have similarities with the capture daemon back-end of GIMS. However, GIMS includes a much broader set of capabilities in terms of the kinds of data that can be collected and how data can be transformed prior to storage.

Two systems that are specifically designed with measurement capabilities for testbed environments include the MINER system [10] and the OMF management framework [25]. The MINER system allows a user to initiate both active and passive measurement tools on testbed hosts. A primary goal of MINER is to provide a unified API for using arbitrary measurement tools. As a result, it requires some code to be written to adapt existing measurement tools to the MINER framework. OMF is more broadly a management and control framework for network testbeds, but includes integrated capabilities to collect and store experiment-specific measurements. GIMS has some similarities with these systems, but is geared specifically toward providing high-performance passive measurement capabilities. It is also control framework agnostic: it can either be run in an independent manner, or easily interface with an existing testbed or network management platform.

6 Summary and Future Work

The ability to capture packets traversing links in a network testbed is of central importance to many different types of experiments. However, the costs and complexity of deploying and managing a packet capture measurement infrastructure within a testbed can be very high. In this paper, we describe a packet capture management environment for network testbeds that we call GIMS. The key objective of GIMS is to reduce the complexities associated with managing and using a distributed packet capture infrastructure.

The design of GIMS is divided into three components. The *front end* includes a set of GUIs that facilitate management and use of distributed packet capture devices. The *back end*, controls configurations of packet measurement and facilitates communication between the front end and the packet capture devices. The *packet capture control system* enforces privacy policies, summarizes and aggregates packet data, and archives data for multiple simultaneous experiments on packet capture devices.

GIMS was developed to run within the GENI infrastructure and has interfaces to directly interact with the ProtoGENI control framework. However, it has also capable of running autonomously outside of ProtoGENI. It was implemented to run on commodity PC hardware with UNIX-based operating systems. It has been running in the WAIL testbed for nearly a year and the software distribution is openly available to the community.

Our on-going efforts are focused in three areas. First, we are in the process of deploying GIMS nodes within the GENI infrastructure. This will enable the system to be used by researchers in that environment. We are also adding additional capabilities for results analysis and reporting. This recognizes the fact that the process of running experiments does not stop with packet capture and requires standardized tools to make use of that data. Finally, to enable the system to be more broadly used, we plan to add new interfaces for other GENI control frameworks such as Orbit [5] and PlanetLab [6].

References

1. Iperf 2.0.5 – the TCP/UDP bandwidth measurement tool (2012), http://iperf.sourceforge.net/
2. Cisco IOS Netflow (2012), http://www.cisco.com/en/US/products/ps6601/products_ios_protocol_group_home.html
3. Endace, Inc. (2012), http://www.endace.com
4. GENI — Global Environment for Network Innovations (2012), http://www.geni.net/
5. The Orbit Testbed (2012), http://www.orbit-lab.org/
6. The Planetlab Testbed (2012), http://www.planet-lab.org/
7. Wireshark — go deep (2012), http://www.wireshark.org/
8. Barford, P.: The Wisconsin Advanced Internet Laboratory (2012), http://groups.geni.net/geni/wiki/MeasurementSystem
9. Barford, P., Sommers, J., Crovella, M.: Instrumentation and Measurement for GENI (2012), http://www.schooner.wail.wisc.edu

10. Brandauer, C., Fichtel, T.: MINER — A measurement infrastructure for network research. In: Proceedings of Tridentcom 2009 (2009)
11. Braun, L., Didebulidze, A., Kammenhuber, N., Carle, G.: Comparing and improving current packet capturing solutions based on commodity hardware. In: Proceedings of ACM Internet Measurement Conference (2010)
12. Brownlee, N.: Using NeTraMet for production traffic measurement. In: Proceedings of IEEE/IFIP International Symposium on Integrated Network Management (May 2001)
13. CAIDA. Coralreef software suite (2012), http://www.caida.org/tools/measurement/coralreef/
14. Cardigliano, A., Deri, L., Gasparakis, J., Fusco, F.: vPFRING: Towards Wire-Speed Network Monitoring using Virtual Machines. In: Proceedings of ACM Internet Measurement Conference (2011)
15. Case, J., Fedor, M., Schoffstall, M., Davin, J.: RFC 1157: A Simple Network Management Protocol (SNMP) (May 1990), http://www.ietf.org/rfc/rfc1157.txt
16. Deri, L.: Improving passive capture: Beyond device polling. In: Proceedings of SANE (2004)
17. Claise, B. (ed.): RFC 3954: Cisco Systems NetFlow Services Export Version 9 (October 2004), http://tools.ietf.org/html/rfc3954
18. Barford, P. (ed.): GENI Instrumentation and Measurement Systems (GIMS) Specification, GDD-06-012 (2006), http://groups.geni.net/geni/wiki/GeniInstMeas
19. Fan, J., Xu, J., Ammar, M., Moon, S.: Prefix-Preserving IP Address Anonymization. Computer Networks 48(2) (October 2004)
20. Gonzalez, J.M., Paxson, V.: Pktd: A packet capture and injection daemon. In: Proceedings of Passive and Active Measurement Workshop (2003)
21. Hussain, A., Bartlett, G., Pryadkin, Y., Heidemann, J., Papadopoulos, C., Bannister, J.: Experiences with a continuous network tracing infrastructure. In: Proceedings of the ACM SIGCOMM Workshop on Mining Network Data (2005)
22. Sommers, J., Barford, P., Duffield, N., Ron, A.: Multi-objective monitoring for sla compliance. IEEE/ACM Transactions on Networking 18(2) (2009)
23. Jacobson, V., Leres, C., McCanne, S., et al.: Tcpdump (1989), http://www.tcpdump.org/
24. Oetiker, T.: MRTG: The Multi Router Traffic Grapher. In: Proceedings of USENIX LISA 1998 (1998)
25. Rakotoarivelo, T., Ott, M., Jourjon, G., Seskar, I.: OMF: a control and management framework for networking testbeds. ACM SIGOPS Operating Systems Review 43(4) (2010)
26. Sadasivan, G., Brownlee, N., Claise, B., Quittek, J.: RFC 5470: Architecture for IP Flow Information Export (March 2009), http://tools.ietf.org/html/rfc5470
27. Sommers, J., Barford, P.: Self-configuring network traffic generation. In: ACM SIGCOMM Internet Measurement Conference (October 2004)
28. Veitch, D., Babu, S., Pasztor, A.: Robust Synchronization of Software Clocks Across the Internet. In: Proceedings of ACM SIGMETRICS (2004)