Scalable Integrated Performance Analysis of Multi-Gigabit Networks

Ezra Kissel*, Ahmed El-Hassany†, Guilherme Fernandes†, Martin Swany†,
Dan Gunter‡, Taghrid Samak‡, Jennifer M. Schopf§
* School of Computer and Information Sciences, University of Delaware, Newark, DE 19716
† School of Informatics and Computing, Indiana University, Bloomington, IN 47405
‡ Computational Research Division, Lawrence Berkeley National Laboratory, Berkeley, CA 94720
§ Informatics Group, Woods Hole Oceanographic Institution, Woods Hole, MA 02543

***Abstract*—**
Monitoring and managing multi-gigabit networks requires dynamic adaptation to end-to-end performance characteristics. This paper presents a measurement collection and analysis framework that automates the troubleshooting of end-to-end network bottlenecks. We integrate real-time host, application, and network measurements with a common representation (compatible with perfSONAR) within a flexible and scalable architecture. Our measurement architecture is supported by a light-weight eXtensible Session Protocol (XSP), which enables context-sensitive adaptive measurement collection. We evaluate the ability of our system to analyze and detect bottleneck conditions over a series of high-speed and I/O intensive bulk data transfer experiments and find that the overhead of the system is very low and that we are able to detect and understand a variety of bottlenecks.

I. Introduction

Understanding bulk data transfer performance is an ongoing problem both for end users and network administrators. It is often impossible to tell what the behavior of a file transfer should be, or where the performance bottleneck in the route lies. Assumptions are often made about network behavior, only to discover after lengthy investigation and wasted tuning efforts that the problem is really in disk or CPU performance.

This paper describes a methodology for collecting a broad set of measurements using a flexible architecture in order to understand system performance across the full end-to-end path. The rate that measurements are taken can be adapted to capture unusual behavior. By avoiding collecting every metric possible at all times, we can scale the number of measurements taken as appropriate for the activity and avoid capturing data when it isn't needed. We incorporate summarization into the information providers collecting the measurements, which also increases the scalability. Our experiments show that the overhead of our approach is low enough that it can be used even with 100Gb/s networks.

Our methodology leverages several tools to provide a user-centric view of distributed system performance: perfSONAR [11], which provides a distributed measurement reporting framework; NetLogger [18], which enables distributed event tracing; NetLogger Calipers, a new component which enables non-intrusive instrumentation and summary statistics for high-frequency measurements; and BLiPP, a flexible framework for collecting host metrics. These components interact using the eXtensible Session Protocol (XSP) [17], a Session Layer protocol that provides transport connections for the exchange of measurement data and the ability to control measurement activities and frequency.

The main contributions of this paper include: An approach to summarized logging that can scale to 100Gb/s networking measurements for use in bulk file transfer behavior tracking; The novel use of XSP to not only communicate data out to an archival system but to communicate control data back in to an information provider to adjust monitoring rates; and a completely general framework that allows end users or networking engineers to include additional data and metadata into an overall logging system. Our analysis focuses on a bottleneck detection methodology that relies on application and host metrics collected by information providers within our system.

II. Background

This section briefly describes existing technologies used in our measurement architecture.

A. perfSONAR

perfSONAR is a web services-based framework that enables network performance information to be gathered and exchanged in a multi-domain, federated environment. perfSONAR has been designed to accommodate easy extensibility for new network metrics and to facilitate the automatic processing of these metrics as much as possible via a flexible schema and data model [25]. While perfSONAR is currently focused on publishing network metrics, it is designed to be able to work with data from any information provider. The global perfSONAR community has many participants, and the current list is available from the perfSONAR web site [4].

B. NetLogger

The NetLogger Toolkit [22] provides an integrated set of tools to collect, archive, and analyze sensor data. Distributed applications can be modified to include NetLogger calls to enable time-stamped traces of events in order to understand performance over many elements. The format and structure of the traces from NetLogger are defined in the Logging Best Practices (BP) document [14]. The BP and perfSONAR

schema are compatible, i.e. BP logs can easily be transformed to perfSONAR data and vice-versa.

C. eXtensible Session Protocol (XSP)

XSP is a Session Layer protocol that was developed for improving network control [17] and has been previously used to enable performance proxies [18]. XSP enables applications to establish a network path that can be used to communicate on-demand setup and configuration of services and communication of application data via a control channel or a separate data channel.

XSP replaces calls to the socket library and adds an explicit session layer by providing protocol encapsulation of the underlying transport layer. For example, when an application calls `connect()`, the XSP library connects to an intermediate service, in our implementation, a daemon called XSPd.

III. System Description

Like most monitoring systems, ours, shown in Figure 1Overall system architecturefigure.1, consists of a set of information providers (for example NetLogger Calipers), that communicate to an archival store. What is novel about our approach is that our information providers communicate using the extensible Session Protocol (XSP) to initiate and control the monitoring activities. In this manner we can adapt the monitoring frequency of the measurement streams and increase or decrease monitoring as appropriate, which allows our system to be highly scalable.

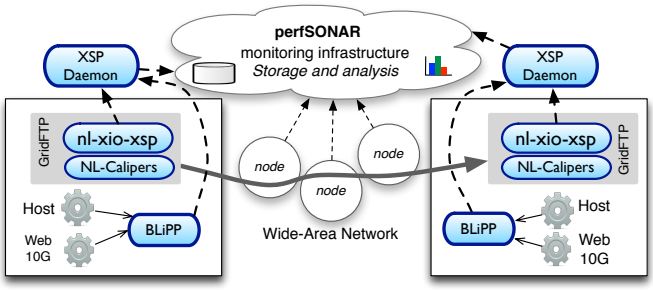

Fig. 1. Overall system architecture

A. XSP for Measurements

We use XSP for the transport of measurement data, to turn on and off information providers, and to change the frequency of collected events. User applications may integrate the XSP client library directly to coordinate with other collection services or expose particular metrics of interest.

An XSP *session* is a time period of interest defined by the application and managed by the XSP control channel. Typical time periods of interest are a series of file transfers or the duration of a virtual circuit. A *context* is the set of measurements and related system information that are collected during a session. A user application will generally generate many sessions, each with its own context. Different sessions can have different types of data in their associated context and potentially different collection rates. The XSP daemon aggregates the session data from multiple information providers into a context, which is then forwarded to the archival store for analysis. The context information can also be used to correlate related measurements from other information providers.

B. Measurement Schema

The existing perfSONAR system exchanges data as XML documents, using Web-Services (SOAP) [25]. While standard, these technologies are very verbose and expensive to parse and generate. One of the focii of this work is to use lightweight and efficient technologies to reduce overhead. Therefore, our monitoring components use XSP as a transport instead of SOAP with a more efficient implementation of the perfSONAR schema. Rather than XML documents, we represent sets of measurement data and metadata in a binary encoding of the Javascript Standard Object Notation (JSON) [2], called BSON [8].

The BSON schema retains the essence of the perfSONAR schema, which explicitly separates *metadata* and *data* for a time-series of values. The metadata items include a globally unique identifier and event type (what is being measured), and may include time-series parameters such as the start & end time and the sampling interval, or other descriptive fields about the data. We enable the re-use of common metadata, such as host name and operating system, by allowing metadata items to point to parent metadata.

The data items have a minimal representation with two components: a metadata identifier and a list of timestamp/value pairs. A key to efficient transmission of measurements is that the metadata needs to be sent only once, before or along with the first set of measurements, and then subsequent measurements send only the data items. The metadata items are retained by an XSP receiver (e.g. XSPd), as part of the context of the session. The BSON schema also allows arbitrary grouping of measurements for fine-grained control of the latency/efficiency tradeoffs. A typical usage would be to place all the measurements from one session in a single BSON structure.

The BSON schema remains compatible with existing perfSONAR schemas, and measurements can be translated from one to the other. Our architecture is not tied to any particular measurement schema, and thus could also use the existing perfSONAR XML if necessary.

C. BLiPP

We gather the needed host data for our system using the Basic Lightweight PerfSONAR Probe (BLiPP). BLiPP reads values from the operating system as directed through configuration and signals from XSP. For example, on Linux systems this involves reading from the `/proc` file system to report relevant system variables. BLiPP can also interface to measurement tools deployed on a host, such as Web10G [23]. BLiPP is flexible and lightweight, and additional measurement functionality can be added as appropriate for a given system. Common metrics gathered by BLIPP include host name and IP

address, CPU time, network I/O statistics, and Web10G TCP statistics.

The integration of XSP into BLiPP enables it to adapt its measurement collection to dynamic system state. An application can actively or passively generate XSP signals to have BLiPP begin, end, or modify its measurements. For example, BLiPP uses connection open/close signals to control which TCP connections it measures with Web10G. An example sequence of messages between XIO-XSP, XSPd, and BLiPP is shown in the swimlane diagram in Figure 2Message sequence for dynamic triggering of BLiPP's Web10G monitoringfigure.2. First, XIO-XSP and BLiPP initiate a session with XSPd. Then each begins sending their monitoring data (`OPT_XIO` and `OPT_BLIPP`). Then XIO-XSP sends connection metadata in an `OPT_XIO` message to BLiPP, which triggers the additional Web10G monitoring of those connections for the duration of the session.

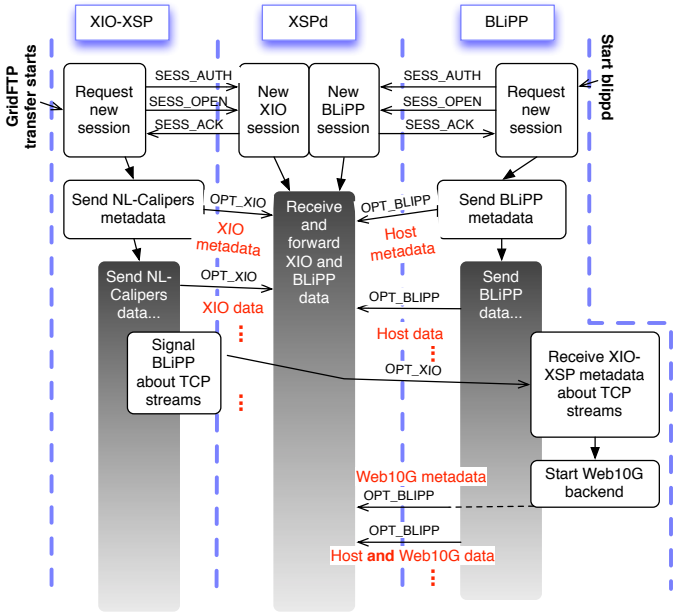

Fig. 2. Message sequence for dynamic triggering of BLiPP's Web10G monitoring.

D. NetLogger Calipers

We have implemented a log summarization extension to NetLogger called the NetLogger *calipers* API, or NL-Calipers, which is a user-level C library that measures the duration of an I/O operation and associates that duration with the number of bytes read or written, the I/O resource, and the context for the session. The log summarization streamlines our earlier work [15], to reduce overhead at high frequencies, such as those seen for 10Gb/s and 40Gb/s networks. The key concept of the approach is that the data collected for each I/O operation is not stored, but instead a set of in-memory summary statistics is updated. This minimizes the processing time and the memory footprint.

Each system call has an associated duration, dur (the end time minus the start time), and number of bytes, $nbytes$, read or written. The summary statistics kept by three NL-Calipers are a five-tuple $\langle$min, max, sum, mean, stdev$\rangle$ for values for dur, $nbytes$, and the ratio $nbytes/dur$. Our experiments show that this five-tuple is sufficient for bottleneck determination, but future work may extend the tuple to include other summary statistics such as histograms or quantiles. We also note that the NL-Calipers library is not specific to I/O instrumentation and may be used to efficiently summarize any high-frequency timeseries.

E. Instrumenting GridFTP / XIO

The well-known file transfer application GridFTP [16] within the Globus Toolkit [12] uses a modular, extensible input/output framework called XIO [16]. This modularization allowed us to insert a *xio-nl-xsp* instrumentation layer, known as a *driver* in XIO, that leverages both our NL-Calipers and XSP libraries. We are thus able to collect and report measurements on the underlying file and network I/O operations. Note that the network I/O includes any protocol supported by GridFTP, including TCP, UDT [7], and Phoebus [19].

Within the xio-nl-xsp driver, NL-Calipers is used to instrument the system calls for the disk and network. This allows us to report the time spent blocked and number of bytes transferred. From this data we generate a time-series of transfer rates. The xio-nl-xsp driver provides functions for reporting these summary statistics using our perfSONAR schema over XSP. In addition to NL-Calipers data, the driver adds transfer metadata such as source IP/ports, destination IP/ports, parallel stream identifier, filename, and file size.

IV. Experimental Evaluation

This section presents the results of the evaluation of our system components. In particular, we focus on our ability to collect and analyze application and host metrics to detect bottleneck conditions that may occur during bulk data transfers.

A pair of hosts built with off-the-shelf components, H_A and H_B, were used in the experiments. Both hosts have a hex-core Intel Core™ i7 980X 3.33GHz CPU, Myricom 10G-PCIE2-8B2-2S NIC, 3ware 9750-8i hardware RAID, and 12GB of memory. The major difference between them is that H_A has 16x500GB SAS drives compared with 2x500GB SAS drives in H_B. This creates a natural I/O imbalance that is revealed in our experiments.

A. Instrumentation overhead

It is important to understand the potential for the instrumentation to skew the measurements. Our first set of experiments measured the instrumentation overhead, and the results show that at the I/O rates used for the rest of the experiments, this overhead is negligible. These results were collected from a microbenchmark that involved instrumenting a loop that performed calculations over an array. The overhead was determined by subtracting the NL-Calipers measured time from the overall time for the operation. NL-Calipers summaries were reported once per second, giving 60 data points, and the microbenchmark was run for one minute on both H_A and H_B.

Because we are interested in steady-state behavior, we used the 10% trimmed mean to estimate the overhead. The results were: $H_A = 83ns/iteration$ and $H_B = 120ns/iteration$. We will use, below, the pooled 10% trimmed mean of all 120 measurements, which is $103\ ns/iteration$.

To understand this overhead when applied to a real-world application, we assume a block size of 256KB, which is the current default used in GridFTP. At 10Gb/s there are 4882.8125 256KB data blocks per second. As there are two NL-Calipers-wrapped system calls per data block at each host (disk and network), we estimate the overhead at 10Gb/s as:

$$\frac{2\ calls}{block} * \frac{103ns}{call} * \frac{4882.8125\ blocks}{1\ second} = \frac{502929ns}{1\ second} \approx 0.05\%$$

Thus, we estimate the overhead as 0.05% at 10Gb/s rates, 0.5% at 100Gb/s rates, and 5% at 1000Gb/s transfer rates.

Overhead of a loop with no I/O is a worst-case scenario since modern systems effectively overlap computation and I/O. We measured the overhead of disk-to-disk (D2D) and memory-to-memory (M2M) transfers from H_B to H_A. Transfers were performed with GridFTP `globus-url-copy`, both instrumented and un-instrumented. To skip TCP ramp-up, we took 5 samples of average throughput starting ≈ 30 seconds into the transfer. Un-instrumented D2D transfers had throughput with mean and standard deviation of 850 ± 10Mb/s, and M2M had 9437 ± 4Mb/s. Mean instrumented D2D throughput was slightly *faster* at 851Mb/s and mean M2M throughput was within 1Mb/s. Thus, for a real transfer at up to 10Gb/s, the overhead caused by instrumentation was much smaller than the random variation in transfer throughput.

In summary, we estimate that on current systems NL-Calipers has $< 0.05\%$ overhead at up to 100Gb/s and should have only $< 5\%$ overhead at up to 1000Gb/s.

B. System Benchmark

Our next set of experiments benchmark the network and disks used in our test configuration. The results, shown in Table IDisk and network benchmark resultstable.1, establish a point of reference for our later analysis. For the disk benchmark, we used the standard *iozone3* [1] tool with options `iozone -s48G -r256k -i0 -i 1-l1 -u4 -F<files>`, to run sequential read and write tests for 48GB of data using 256KB blocks and from one to four concurrent processes. Our results show that multiple concurrent processes usually degrade throughput, but H_A write throughput almost doubles from 1 to 4 writers, likely due to better load-balancing across all 16 disks.

For the network benchmark, we tuned the test system to have the maximum TCP buffers set to 256MB, metric caching disabled, autotuning enabled, and the default congestion control algorithm selected in Linux kernel 2.6.35 set to CUBIC. We used the netem [3] kernel module to emulate wide-area network (WAN) latency of 100ms. We then ran a series of GridFTP tests. Each test was run using both one and four parallel streams.

Disk

Host	Processes	Read (Gb/s)	Write (Gb/s)
H_A	1 / 4	11.2 / 0.8	4.8 / 8.8
H_B	1 / 4	0.9 / 0.5	0.8 / 0.8

Network

TCP/UDT	Latency	Streams	Throughput (Gb/s)
TCP	0ms	1 / 4	9.5 / 9.5
TCP	100ms	1 / 4	8.0 / 9.4
UDT	0ms	1 / 4	4.1 / 4.0
UDT	100ms	1 / 4	0.8 / 3.2

TABLE I
DISK AND NETWORK BENCHMARK RESULTS

C. Bottleneck Analysis

Our final set of experiments demonstrate use of the application and system-level measurements for bottleneck analysis. Using the same experimental setup as above, we performed *instrumented* GridFTP transfers from H_B to H_A. These transfers were memory-to-memory, memory-to-disk, and disk-to-disk while varying the transfer protocol (TCP and UDT), the netem-simulated latency ($0ms$ and $100ms$), and the number of parallel streams. In some experiments we also introduced 0.01% packet loss during the transfer.

Based on the benchmark results, we would expect that over TCP, the bottleneck would be the disk read for disk-to-disk and the disk write for memory-to-disk. For UDT, the network would be the bottleneck except for disk-to-disk transfers. The network would be expected to be the bottleneck for all memory-to-memory transfers.

The three potential bottleneck components are disk read, disk write, or network read. To estimate each component's relative performance, we use the mean($nbytes/dur$) value from NL-Calipers, called *instantaneous throughput* or $tput_i$. The $tput_i$ metric is useful because it only includes the time the component is performing I/O, not time waiting for more data to read or write.

We found that a very simple bottleneck algorithm yielded correct results in most circumstances. For each experiment, we compare the mean $tput_i$ for disk read μ_{dr}, disk write μ_{dw}, and network read μ_{nr}. The lowest value is chosen as the potential bottleneck. A *standardized t-test* is then used to determine whether the difference in mean values between the chosen value and the other two is statistically significant. For example, if μ_{nr} is the lowest value then network read is the potential bottleneck and we perform t-tests for the hypotheses: $\mu_{nr} < \mu_{dr}$ and $\mu_{nr} < \mu_{dw}$. If this test fails, then the bottleneck may be in any of the statistically similar components.

Figure 3Time series of $tput_i$ for representative TCP experiments: (a) 1 stream memory-to-disk with 100ms latency, (b) 1 stream memory-to-memory with no latency, (c) 1 stream disk-to-disk with no latency, (d) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 secondsfigure.3 shows plots of the average $tput_i$ over time for four TCP experiments. These plots show some of the different behavior patterns that can be detected from the application-level information.

The memory-to-disk transfer in plot (b) reflects a network bottleneck where the "disk" is effectively infinitely fast. The disk-to-disk transfer in plot (c) shows, as the benchmarks indicated, that the disk read is slower than either the disk write or the network.

Figure 3Time series of $tput_i$ for representative TCP experiments: (a) 1 stream memory-to-disk with 100ms latency, (b) 1 stream memory-to-memory with no latency, (c) 1 stream disk-to-disk with no latency, (d) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 secondsfigure.3(a) shows that the algorithm detects the network as the bottleneck, even though the benchmarks indicate that for a single stream the disk write should be about 40% slower. This may be due to the xio-nl-xsp instrumentation observing the effects of OS buffering, making the disk write calls within GridFTP appear faster than the actual system write to disk. Future work will explore two possible solutions to this: filtering the timings to be more robust to buffering effects and using host-system (BLiPP) I/O measurements in the bottleneck determination algorithm.

Finally, Figure 3Time series of $tput_i$ for representative TCP experiments: (a) 1 stream memory-to-disk with 100ms latency, (b) 1 stream memory-to-memory with no latency, (c) 1 stream disk-to-disk with no latency, (d) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 secondsfigure.3(d) shows the ability of the application-level monitoring to quickly detect a change event, in this case the introduction of 0.01% packet loss (via netem). Before the loss is introduced the disk write is the bottleneck, as expected in a memory-to-disk transfer. After the loss is introduced, the network throughput begins to degrade. Relatively quickly, the network throughput dips below the disk write throughput and becomes the bottleneck.

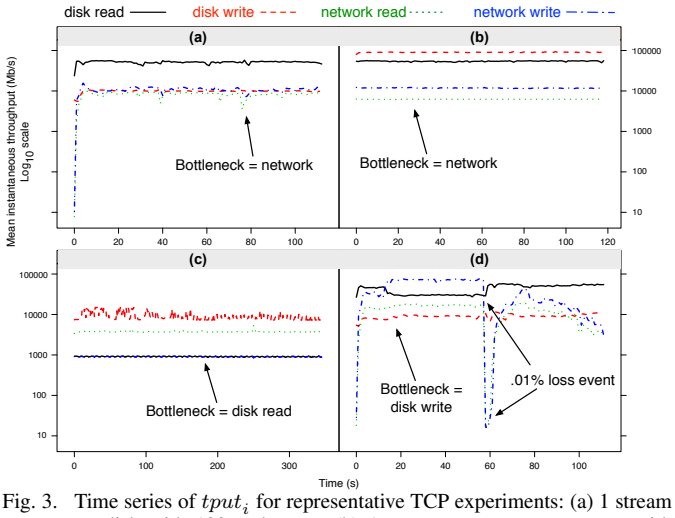

Fig. 3. Time series of $tput_i$ for representative TCP experiments: (a) 1 stream memory-to-disk with 100ms latency, (b) 1 stream memory-to-memory with no latency, (c) 1 stream disk-to-disk with no latency, (d) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 seconds.

Example UDT results are shown in Figure 4Time series of $tput_i$ for representative UDT experiments: (a) 4 streams

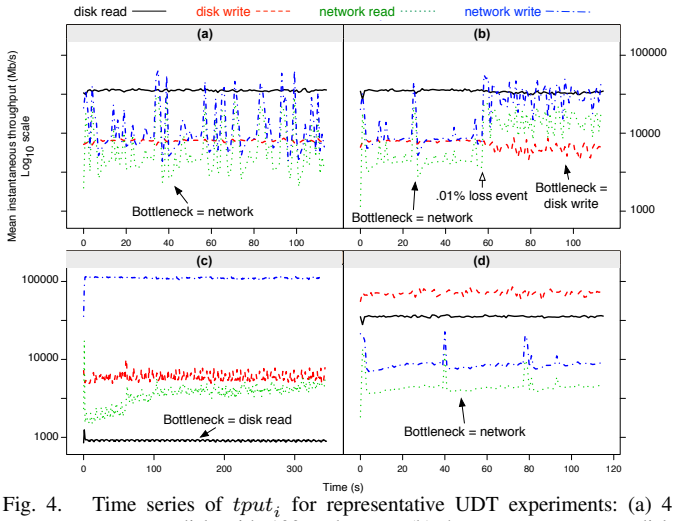

Fig. 4. Time series of $tput_i$ for representative UDT experiments: (a) 4 streams memory-to-disk with 100ms latency, (b) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 seconds, (c) 4 streams disk-to-disk with 100ms latency, (d) 4 streams memory-to-memory with 100ms latency.

memory-to-disk with 100ms latency, (b) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 seconds, (c) 4 streams disk-to-disk with 100ms latency, (d) 4 streams memory-to-memory with 100ms latencyfigure.4. In plots (a) and (d), the network is identified as the bottleneck, as expected. We note, however, the much higher variability in the network reads in (a) when writing to disk than in (d) when writing to memory. Disk reads are the bottleneck, again as expected, for the disk-to-disk transfer shown in plot (c).

Figure 4Time series of $tput_i$ for representative UDT experiments: (a) 4 streams memory-to-disk with 100ms latency, (b) 4 streams memory-to-disk with 100ms latency and 1% loss added at 60 seconds, (c) 4 streams disk-to-disk with 100ms latency, (d) 4 streams memory-to-memory with 100ms latencyfigure.4(b) shows the bottleneck changing from network to disk after the introduction of 0.01% packet loss. This is incorrect: as one would expect, end-to-end throughput dropped with the added packet loss. However, the plot clearly shows that the $tput_i$ for network reads increased dramatically (thus causing the mistaken change in bottleneck).

The increase in $tput_i$ is a side-effect of the instrumentation of the UDT API. For UDT, we have wrapped NL-Calipers around the user-level UDT read, not the system `read()` call. The user-level UDT read will not return any data until all required packets have been buffered in order. Therefore, packet loss will cause some UDT reads to return no data at all (and thus be ignored by NL-Calipers), and others to return much more quickly with a full buffer. This effectively hides I/O time from the instrumentation, skewing $tput_i$ upwards. To verify this, we use the additional application and system values (from NL-Calipers and BLiPP), shown in Figure 5Timeseries for a UDT transfer with 0.01% loss introduced at 60 seconds: (a) Count of reads per 1-second sampling interval, (b) Standard deviation of $tput_i$, (c) User and system CPU usagefigure.5, for

this transfer. Plot (a) shows that, after loss starts at 60 seconds, the number of successful UDT reads decreases by about half and plot (b) shows that the variance in $tput_i$ of reads almost triples. This corresponds to fewer successful reads, some of which are much faster. This trend is also evident in the CPU load values shown in plot (c), where less work is being done at both the user and system level.

In future work, we will modify our bottleneck algorithm to use the change in variance as well as mean values for $tput_i$.

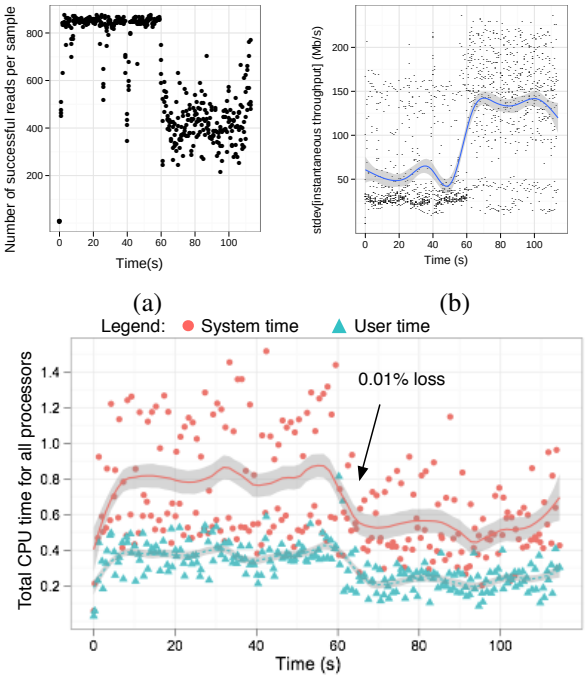

Fig. 5. Timeseries for a UDT transfer with 0.01% loss introduced at 60 seconds: (a) Count of reads per 1-second sampling interval, (b) Standard deviation of $tput_i$, (c) User and system CPU usage.

Table IIBottleneck analysis resultstable.2 summarizes the results of our bottleneck detection method for each experiment configuration. All memory-to-memory variants showed the network as the bottleneck and are thus not shown in the table.

Resources	Protocol	Streams	Latency / Loss 0 / 0	100ms / 0	100ms / 0.01
Disk to Disk	TCP	1	disk.**read**	**network**	n/a
		4	disk.**read**	**network**	n/a
	UDT	1	disk.**read**	disk.**read**	n/a
		4	disk.**read**	disk.**read**	n/a
Memory to Disk	TCP	1	**network**	**network**	**network**
		4	disk.**write**	disk.**write**	disk.**write**
	UDT	1	**network**	**network**	**network**
		4	**network**	**network**	disk.**write**

TABLE II
BOTTLENECK ANALYSIS RESULTS

Many other improvements could be made to this algorithm, including change-point detection [9] and recognizing resource contention; these are made possible by our detailed and correlated host and application-layer measurements.

V. Related Work

There is considerable work done in end-to-end performance debugging and optimization. Distributed systems tracers [6], [10], [11], [20] modify the protocols, libraries, and/or middleware used by applications to log the progression of requests as they traverse the multiple layers of the system. These approaches provide a very detailed view of end-to-end performance, but require ubiquitous deployment over the end-to-end path (usually within a datacenter). In contrast, the work presented in this paper leverages the widely deployed perfSONAR monitoring infrastructure by adding host and application metrics under the same representation to create an end-to-end performance model. There is also little known about how tracers compose with standard network performance measurements widely used by network administrators (e.g. SNMP counters). The ability of our framework to start new measurements or adjust the rate of ongoing measurements on demand implements some of the "reactive measurement" concepts described in [5].

Network bottleneck analysis has also been addressed in previous work, with Sun et. al. [21] and SNAP [24] being most closely related to our own work. Both approaches use TCP statistics to infer if bottlenecks arise at the application, the network stack in the host, or the network path. SNAP also collects socket-call logs to record the time and number of bytes read/written by a socket syscall. We improve upon these approaches by adding application defined measurements, which lets us directly pinpoint the source of bottlenecks within the application and not only infer the problem from the TCP statistics. BLiPP, presented in this paper, also collects TCP statistics if Web10G [23] is available. Our work is also, to the best of our knowledge, the first to scale with low overhead to 10G data transfers and beyond.

VI. Conclusion and Future Work

We have outlined an architecture for a flexible, scalable, and integrated instrumentation and measurement system. A number of key system components have been developed and benchmarked and we have evaluated our approach in detecting bottleneck conditions across various bulk-data transfer scenarios. Our preliminary analysis demonstrates the ability to correctly aggregate and correlate collected metrics and accurately determine the cause of end-to-end performance bottlenecks. As future work, we plan to extend the analysis capability within our architecture and enable closer integration with external measurement sources.

Acknowledgments

This work was supported in part by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under contracts DE-AC03-76SF00098 and DE-AC02-06CH11357, as well as by the National Science Foundation under grant OCI-0943705.