Log Summarization and Anomaly Detection for Troubleshooting Distributed Systems

Dan Gunter [#1], Brian L. Tierney [#2], Aaron Brown [*3], Martin Swany [*4], John Bresnahan [!5], Jennifer M. Schopf [!6]

[#] *Lawrence Berkeley National Laboratory, Berkeley, CA, USA*
[1] dkgunter@lbl.gov
[2] bltierney@lbl.gov

[*] *University of Delaware, Newark, DE, USA*
[3] brown@cis.udel.edu
[4] swany@cis.udel.edu

[!] *Argonne National Laboratory, Argonne, IL, USA*
[5] bresnaha@mcs.anl.gov
[6] jms@mcs.anl.gov

***Abstract*— Today's system monitoring tools are capable of detecting system failures such as host failures, OS errors, and network partitions in near-real time. Unfortunately, the same cannot yet be said of the end-to-end distributed software stack. Any given action, for example, reliably transferring a directory of files, can involve a wide range of complex and interrelated actions across multiple pieces of software: checking user certificates and permissions, getting details for all files, performing third-party transfers, understanding re-try policy decisions, etc. We present an infrastructure for troubleshooting complex middleware, a general purpose technique for configurable log summarization, and an anomaly detection technique that works in near-real time on running Grid middleware. We present results gathered using this infrastructure from instrumented Grid middleware and applications running on the Emulab testbed. From these results, we analyze the effectiveness of several algorithms at accurately detecting a variety of performance anomalies.**

I. Introduction

Many of today's Grids have ongoing performance and reliability problems that have yet to be addressed. Grid2003, now Open Science Grid (OSG) [1], saw a 30% job submission failure rate[2], with 90% of the failures caused by problems such as disk filling errors, gatekeeper overloading, and network interruptions. This error rate has reduced in the past three years to around 15%, with the current goal of attaining a 90% success rate this year. Yet clearly having one of every ten job submissions fail is not acceptable in a production setting. Similar results have been seen on other Grids and Grid testbeds. For example the GrADS testbed [3] found that 20% of runs failed due to NFS problems at one site [4]. These data points only mention hard failures. The numbers for soft failures (performance reductions) may well be much higher, and likely go unnoticed.

Troubleshooting Grid middleware is made more difficult by the large number of interconnected components. For example, a single action, such as reliably transferring a directory of files, can result in the coordination of a wide suite of loosely coupled software tools. These include security software to handle the certificates, check permissions, perform delegation, and possibly encrypt the message streams, file transfer tools to check the disk space, set up the connections, and transfer data between resources, and reliability software that must understand re-try policies, track transfer status behavior, and react to failures. Each of these systems may suffer from various forms of failure, which may or may not be reported. If failures are reported, it is typically via a log file with various styles of logging. Combining log information from several components in order to understand what caused a given failure can be challenging. However this is exactly what is needed to troubleshoot a problem as it cascades from one component into the next.

Sufficiently detailed log data is often not available in any form. For real-time debugging of interleaved and interrelated software stacks, we often need a function-level execution trace, but logging at that level of detail can easily become unmanageable. For example, a full trace of the I/O operations performed by a single GridFTP server [5] capable of saturating a 10 Gigabit network will generate O(20,000) log events per second, or over 70 million per hour. If Grid middleware components generally ran this level of detailed monitoring, the perturbation would be unacceptable. And yet logging only coarse-grained information and asynchronous status messages, as is the common practice today, makes debugging failures a heroic effort.

Due to its complexity and heterogeneity, Grid middleware lags behind stand-alone system tools in terms of anomaly detection, where an *anomaly* is defined as an unexpected degradation in behavior that adversely affects application performance. Detecting failures is made more difficult by fault tolerance mechanisms, such as retries in GridFTP transfers and most workflow engines, which may mask more serious errors. Detecting performance degradation is also complicated by high system variance and periodic patterns that last days or weeks.

The contributions of this paper include:

- A general purpose technique for configurable summarization of time-series data that scales to high event rates, detailed in Section III.

- An anomaly detection technique that and can adapt to changing performance baselines, detailed in Section IV.

An additional contribution of this paper is the description of an scalable infrastructure for collecting and normalizing log data for troubleshooting of complex distributed systems, detailed in Section II.

Our anomaly detector is targeted at middleware and applications with consistent expected performance. This includes large file transfers, predictable user computations, and many streaming data analysis and visualization services, all common on today's Grids.

We show experimental results for our log summarization demonstrating it can provide detailed real-time information without perturbing the systems being monitored.

We also explore several anomaly detection methods, and show that in our experiments no single method was optimal. In practical deployments a collection of anomaly detection methods is likely the best strategy.

While there are several projects related to distributed system performance monitoring and anomaly detection such as Pablo [6], TAU [7], and Paradyn [8], none of these projects are focused on the near real-time detection of performance problems that are due to transient system and middleware performance faults. We review related work in more detail in Section VII.

II. Distributed Troubleshooting Infrastructure

Our log summarization and anomaly detection components must, to be useful, integrate into an overall Grid troubleshooting infrastructure. This section lays out the components needed for end-to-end distributed systems troubleshooting.

A. Background

We assume an environment in which the monitored components, including long-running services and applications, are managed under separate administrative domains. For security, privacy, and manageability reasons, the logs of these components may only be partially shared off-site. Networks between the monitored components and the outside world may also be slow, unreliable, or both.

An example of one such large distributed system is the Open Science Grid (OSG) [1], a Grid infrastructure currently comprised of over 75 international sites. Services can have uptimes measured in days or weeks, and application developers regularly use tens of sites at a time. Failure rates can be quite high, and debugging a system problem can take days due to inaccessible logs or lack of detailed information.

B. Approach

We are working with the Open Science Grid to define, implement, and deploy this troubleshooting infrastructure. The key elements of our approach are to instrument applications following a set of logging best practices, aggregate and filter logs with *syslog-ng* (described below), and then analyze the streaming data while archiving it to a relational database. We believe this provides a logging infrastructure that allows for scalable analysis of the distributed logs. Each component is described in more detail below.

In logging, the old adage applies: Garbage In, Garbage Out. The best analysis system in the world cannot do much with bad input. So, we have written a recommendations guide for Logging Best Practices [9] that combines good instrumentation practices with log format guidelines.

- Practices. All logs should contain a unique *event* name and an ISO-format timestamp [10]. All system operations that might fail or experience performance variations should be wrapped with `start` and `end` events. All logs from a given execution thread must be tagged with a *globally unique ID* (or `GUID`) [11], such as a Universal Unique Identifier (UUID) [12] that allows a set of related logs from a given execution to be grouped together.
- Log format. Logs should be composed of lines of ASCII name=value pairs; this format is highly portable, human-readable, and works well with line-oriented tools.

Troubleshooting analysis often requires detailed comparison of distributed logged information. A distributed query across administrative domains can be difficult to deploy in production Grids such as OSG. Instead, we aggregate the data using the open source tool *syslog-ng* [13], which provides us the ability to filter logs based on program name, log level, and even a regular expression on message contents.

Loading a subset of logs into a relational database enables sophisticated data mining. Historical queries can be used to provide baseline performance information, allowing the comparison of current performance against a historical baseline.

Streaming analysis of the data is more amenable to online anomaly detection. *syslog-ng* can redirect a subset of the events (based on program name, etc.) to an analysis engine, such as a simple *missing event detector*. If all important actions are wrapped with start and end events as recommended, then a large class of troubleshooting problems can easily be found by simply looking for missing end events, which can indicate that something failed without generating an error, or that something is taking too long to complete. Baseline performance information is needed to determine how long to wait before deciding that an event is in fact missing.

The infrastructure described so far provides a foundation for higher level troubleshooting tools. We focus on two new components of this infrastructure, the data summarizer and anomaly detector:

- **Data Summarizer**: As described in Section I , detailed logging for many distributed services may result in delivery of many thousands of log events per second. Logging at this level will affect application performance and potentially fill up disk with unneeded log data. It will also put stress on components that process log data, such as an anomaly detection tool. Therefore we have implemented a log summarization module for the NetLogger Toolkit [14] that can reduce the amount of log data generated by several orders of magnitude, while still capturing key information. By using the summarization module sites

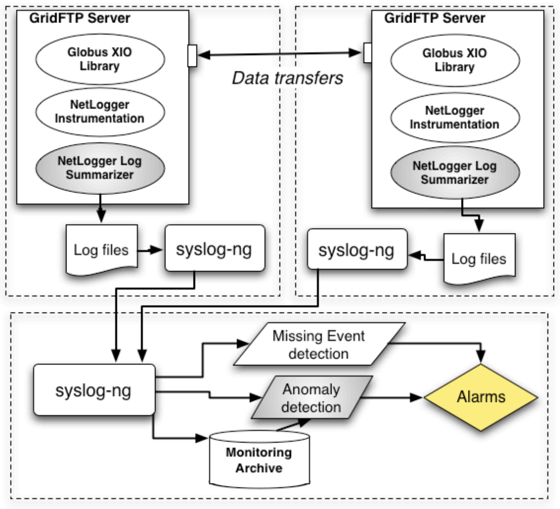

Fig. 1. Log generation and collection process.

will be able to do performance analysis at a very fine granularity and yet scalably collect and manage their logs. This component is described in detail in Section III.

- **Anomaly Detector**: For long-lived services such as large file transfers or long running jobs, we want to be informed of dramatic performance anomalies while the job is still running. For this we have developed an anomaly detection tool that operates on incoming streams of log data, as described in Section IV.

An example of this full system being used for a GridFTP third-party transfer is shown in Figure 1. Logs are summarized, and forwarded to a central location using syslog-ng, where they are analyzed for anomalies.

These components will also form the basis of higher-level services. In general a Grid user or administrator will not wish to know about each and every minor perturbation. They would prefer to specify something like: "Let me know if performance falls below 50% of the expected value for more than 30 minutes". To be useful for both Grid users and administrators, there needs to be an easy way to subscribe to error and anomaly events of interest, and to be able to specify anomaly thresholds. Our longer term vision includes a web services front end to the anomaly detection component that can handle these types of subscription requests.

III. Trace summarization

We have implemented a log summarization extension to the NetLogger Toolkit. Our NetLogger toolkit [14] provides an integrated set of tools for creating sensors and for collecting, archiving, and analyzing sensor data. Using NetLogger, distributed application components are modified to produce timestamped traces of "interesting" events at critical points. This log summarization extension to the NetLogger client library can handle thousands of calls per second without perturbing the application. We present here the underlying model, features, and limitations of the library.

```
for (i=0; i < 1e9; i++) {
  nl_write(log,"foo.start","guid=s",guid);
  double v = foo();
  nl_write(log,"foo.end",
           "guid=s value=d",guid,v); }
```

Fig. 2. Tracing API example

A. Data Model

We model application activity as a time-series of named, timestamped objects we call *events*. In addition to name and timestamp, each event has set of attributes with a primitive numeric or string-valued data value. In this work, we deal with three types of events: markers at the entry or exit to an activity, called respectively *start events* and *end events*; and derived events summarizing a number of start event and end event pairs, called *summary events*. The start and end events must have a GUID (as discussed in Section II.B) so they can be matched. In addition, the end event carries a numeric value (in a user-selected attribute) that we call the *event value*. For example, in the GridFTP end event the event value is the number of bytes read or written, and in a user job the event value may be the number of iterations per unit time. All three event types have a GUID.

B. Summarization Features

Detailed logging is generally avoided in production environments because it can generate too much log data and, at high frequencies, perturb the system. However, log data can be essential for understanding interactions in a complex system. For example, detailed logging of GridFTP events as shown in Table II was needed to debug errors in parallel streaming performance [15]. To solve this dilemma, we have written a logging library that can be configured, at run-time, to produce anything from detailed logs all the way down to a single event per run – for the same logging statement in the code – with virtually no application perturbation at the rate of thousands of log events per second. The API is simple and printf-like, as shown in Figure III-B.

Our implementation of summarization is, currently, tailored to our focus on the start and end of activities (such as loops). The effect is that many repetitions of a pair of start and end events can be compressed into a single derived event. We focus here on summarizing events (of the same type and identity) over a user-provided time period. So, every `N` seconds, all corresponding repetitions of the start and end event pair are combined into one summary event, for which the following statistics are reported (remember, in this model each start/end event pair is also associated with a numeric value):

- min, max, and sum of $event_{end} - event_{start}$
- min, max, and sum of $event\ value$
- sum of $event\ value/(event_{end} - event_{start})$
- last $event_{end}$ - first $event_{start}$

From these statistics, we can easily calculate several additional values, including (a) the proportion of total time was spent in this activity, (b) the rate at which the value changed inside the activity itself (instantaneous rate of change), and (c)

TABLE I

TRACE THROUGHPUT IN EVENTS/SECOND.

Log Destination	Full	Summarized
disk	202,000	588,000
/dev/null	331,000	588,000

the rate at which the value changed during the entire summary period (average rate of change).

C. Log Generation Overhead

Detailed logs have two performance impacts. The first is the performance effects during log generation, primarily perturbation of the process being traced. The second is performance effects after generation, including the time to process, store, or transport the logs themselves. This subsection examines the perturbation caused by log generation, and the following subsection discusses the log management issues.

In order to understand the perturbation caused by logging, we ran a series of tests on an Emulab [16] pc3000 host, which is a 3GHz Intel Xeon, running Linux 2.6.12. The tests were written in C and compiled using gcc 4.0.0 with optimization level -O2. The disk had an average write performance of roughly 60MB/s and only disk writes were performed during the test.

The throughput of the summarized instrumentation using a 1 second summarization interval is shown in Table I. The logs were written to the locally mounted disk or the null device, and each logged event was approximately 100 bytes. The reported number is the mean number of events per second, rounded to the nearest thousand, across 10 runs (the standard deviations were always below 1% of mean).

These results show that summarized logs can be generated more than twice as fast as the raw logs. Summarization is even faster than logging to `/dev/null` because most of the summarized events do not have to be serialized. This is significant, as a quick calculation using the `/dev/null` and disk I/O numbers for the ”full” logging shows that about 60% of the time is spent serializing the event. And because the summarized instrumentation only writes to disk once per second, the I/O performance doesn't change the average summarizer throughput.

From these results, a back-of-the-envelope calculation, assuming that the average latency per call is just the inverse of throughput, gives the summarizer's perturbation of an application that had 10,000 start/end events per second to be 3.4%.

To get a more precise estimate of the amount of perturbation caused by the summarization library, we also ran a microbenchmark with instrumentation of both I/O (disk write) and computations. The microbenchmark had an outer loop that we ran from 200 to 40,000 iterations per second. Each iteration of the loop was wrapped with a start/end event pair. The program first estimated parameters (a fixed number of inner loops) to attain the desired rate of outer loop iterations per second. Then the microbenchmark was run with the same parameters and three different levels of logging: no logging (baseline), full logging, and summarized logging. The results of 10 runs, with the logging levels in random order were averaged (the standard deviation was always below 0.7% of the mean, so is not shown). Percent perturbation was computed as:

$$\frac{Time_{instrumented} - Time_{baseline}}{Time_{baseline}} * 100$$

for both the full and summary logs. The results are shown in Figure 3.

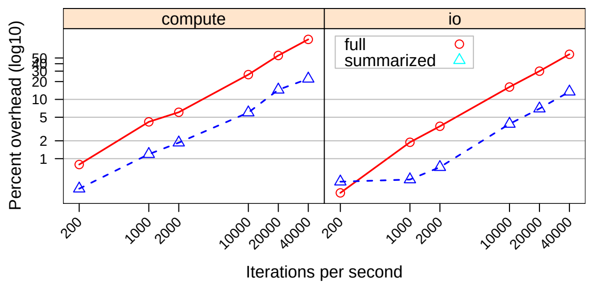

Fig. 3. Perturbation caused by log generation. Full logs and summarization logs compared to a baseline without any logging.

The ratio between the percentage overhead for the full and summarized logs agrees with the log generation throughput measurements (e.g., within $\pm1\%$ of the estimate for 10,000 iterations per second). Since the graph is on roughly a log/log scale, the linear trend in both lines indicates a linear change in perturbation. We note that the summarizer perturbation of the I/O microkernel is roughly flat (at under 1%) up to 2,000 iterations per second, whereas the full logging perturbation of the I/O, and all perturbation of the computational microkernel, increases steadily. This is likely due to overlapping of the computational work of performing the summarization, such as data structure manipulations and computing the statistics, with the disk writes in the I/O microkernel.

Testing this in a more realistic setting, we examined the perturbation of GridFTP servers in Emulab using the default 256KB GridFTP data block size on a 1Gb/s network. The maximum event rate at either server is 1000 start/end event pairs per second, and at this rate the microbenchmark results indicated I/O perturbation to be less than 1%. Independent tests also confirmed that even on LAN networks this much perturbation of the transfer rates was lost in the measurement noise.

D. Log Management Overhead

While it is important that log generation does not perturb the application, it is also important that the resulting data does not overwhelm subsequent processing steps, such as storing the data locally on disk, forwarding it to syslog, or loading it into a relational database. Put another way, you can't just create the data and forget about it; you also need to manage it. Time-based summarization has the property that the volume of log data, given roughly constant sizes for each logged event, depends only on the summarization interval. This makes the log volume much easier to predict, and thus provision for.

IV. Anomaly Detector

Our approach to detecting anomalies is to apply simple and intuitive techniques that are inexpensive enough to run in near real time. Our goal is to provide a system that can provide indications of anomalous behavior in a general case, while supporting customization to allow sites to adapt the sensitivity as needed. Our results show that certain algorithms are more likely to find certain classes of anomalies, and we illustrate a variety of approaches to address this.

A. Anomaly Detection Algorithms

The input to our anomaly detector is a time-series of values from the log summarizer, as discussed in Section III.B. Each value represents the performance of a particular component, e.g., a stream of disk I/O events for a GridFTP transfer or performance data from an inner loop of a computation.

We use two families of anomaly detection algorithms, the mean $\pm N$ standard deviations, *MSD*, and the cumulative distribution function, *CDF*. The CDF family of algorithms is non-parametric and generally more robust to outliers than MSD, since it is simply a step function describing the distribution of a sample. In both cases, the current value of the event stream is compared to a set of *historical data*, and if the current value is above or below a *threshold* then we classify that value as an anomaly.

The historical data set can be formed in a variety of ways as well. We use two different approaches: *Static*, which uses the first N values in a run and does not change over time, and *Heuristic*, which updates the historical data set with the current value if it is not an anomaly, thereby preventing extreme values from being added but still allowing the data set to evolve.

In order to dampen the effect of outliers, we add the option of *smoothing* using the exponential weighted moving average (EWMA) function, which takes a single parameter, α. For each new value, x_i, the smoothed value is calculated as: $\alpha * x_i + (1-\alpha) * x_{i-1}$. Thus, an α of 1 is no smoothing at all, and an α very close to 0 is a large amount of smoothing. The accepted rule-of-thumb value, in absence of further evidence, is to use $\alpha = 0.2$ [17].

We therefore have eight basic algorithms that we compare: MSD Static history Unsmoothed (MSD-su), MSD Static history Smoothed (MSD-ss), MSD Heuristic history Unsmoothed (MDS-hu), and so on for MSD-hs, CDF-su, CDF-ss, CDF-hu, and CDF-hs.

B. Parameters

The *threshold* value determines a distance from a mean or median, outside of which a value is considered an anomaly. Typically, the MSD family uses a number of standard deviations from the mean and the CDF family uses a percent of the tail of the cumulative distribution function. A value of 2 for MSD would correspond to a value of 5% for the CDF, if the data were normally distributed. On real performance data, which is not normally distributed, these settings may yield quite different results.

TABLE II
GridFTP log events.

$sender$	$receiver$
disk.read.start	network.read.start
disk.read.end	network.read.end
network.write.start	disk.write.start
network.write.end	disk.write.end

The *window size* for the heuristic approaches, or the number of initial values for static, determines how much past history to consider when evaluating the current environment.

V. File Transfer Anomaly Experiments

In this section we evaluate which of the anomaly detection methods described in Section IV works best for detecting anomalies in GridFTP performance.

We ran experiments looking for anomalies during GridFTP transfers in Emulab. Data from the log summarization component was used to look for anomalies, and we used a one second summarization interval. The experiments had two settings: a controlled environment where we could easily dictate which periods should be counted as anomalous, and a longer 5 hour run that was more randomized, noisy, and thus realistic.

These experiments show us that for anomalies that affect file transfer performance, smoothing makes a large difference. They also show that the static approach to augmenting the historical data set can easily detect long term shifts in the baseline performance, but that the heuristic approach is better for finding intermittent anomalies.

A. Configuration

We performed bulk data transfers between two hosts using GridFTP. The servers at each host were instrumented with NetLogger so that every network and disk read and write was wrapped with a start and end event. Each 256KB data block transferred from sender to receiver produced the eight log events shown in Table II.

The Emulab testbed, shown in Figure 4, had a single link connecting two routers that allowed us to vary the latency of the connections between two subclusters. Each subcluster had one host running a GridFTP server and two hosts injecting perturbations, as described below. All links were Gigabit Ethernet, and all hosts were Emulab pc3000 machines running a Linux 2.6.19 kernel (which uses BIC TCP). We configured the routers to introduce a one-way latency of 10ms between the GridFTP servers. All file transfers read and wrote from local disk, which we benchmarked at roughly 60MB/s write and 100MB/s read.

Figure 5 shows a typical GridFTP single stream transfer of a 10 GByte file across a 10 ms latency network in Emulab. It shows the typical TCP "sawtooth" performance pattern, peaking around 400 Mbits/s. This plot shows that unperturbed behavior had high short-term variability, so it is important that an anomaly detector not be too sensitive to these local peaks. Although parallel GridFTP, which has smoother behavior, is commonly used for real transfers, we chose to use single

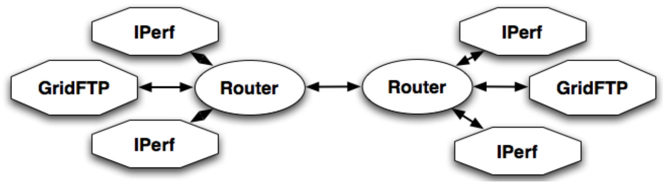

Fig. 4. Emulab configuration.

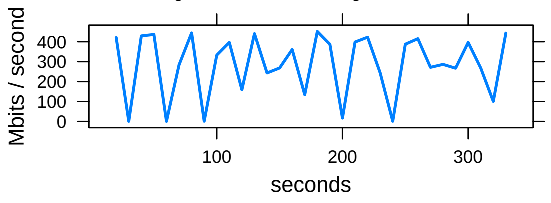

Fig. 5. Typical throughput of a 1 stream GridFTP on 10ms path. Note the typical TCP sawtooth performance pattern, demonstrating that one must be careful to not flag this type of performance variance as an anomaly.

stream GridFTP for these tests. We did so precisely because the performance is more variable, and we wished to ensure that our anomaly detection approach was not too sensitive to these variations.

We set our threshold parameters to the standard 2 standard deviations, for the MSD family, and 5% for the CDF family. For the historical data set we used 1800 initial values for the Static approaches and the full data set for the Heuristic approaches.

B. Injected Perturbations

Anomalies do not occur naturally in our Emulab testbed so we simulate them by injecting perturbations into the system. We injected three types of perturbations:

- A **disk injection** competed with GridFTP for disk I/O resources. We used a simple program based on the Unix `dd` command. The program ran for 2 minutes for the controlled data, and for a random value between 1 second and 1 minute for the 5 hour run.
- A **network injection** competed with GridFTP for network bandwidth by running `iperf` TCP tests. This category is further broken down into one-stream and four-stream network injections, which we will call *weak* and *strong* network injections, respectively. These lasted 5 minutes for the controlled run and a random value between 1 second and 2 minutes for the 5 hour run.
- A **latency perturbation** modified our Emulab network topology to quadruple the latency on the link connecting the GridFTP servers. This had the affect of decreasing the throughput of a one-stream GridFTP transfers by almost 50%.

C. Controlled Data Results

The controlled data run was 90 minutes in length, during which we injected three types of perturbations at 5 minute intervals: strong network, weak network, and I/O. The number of injections is shown in the left column of Table IV. One hour into the run, we injected a latency perturbation, which continued until the end of the run. Figure 6 shows the GridFTP performance during the controlled run, and two different anomaly detection algorithms, MSD-su and CDF-hs. The symbols along the bottom of the plot show the start of various injected performance anomalies. Note that the MSD-su detects the latency shift, unlike the CDF approach, but even by eye has a much higher false positive rate.

TABLE III
ANOMALIES DETECTED IN THE CONTROLLED RUN FOR THE EIGHT ALGORITHMS.

Algorithm	Correct	FP	FN	Badness
MSD-ss	9	4	0	0.44
MSD-su	9	3	0	0.33
MSD-hs	**8**	**0**	**1**	**0.125**
MSD-hu	7	0	2	0.29
CDF-ss	9	4	0	0.44
CDF-su	9	3	0	0.33
CDF-hs	**8**	**0**	**1**	**0.125**
CDF-hu	5	0	4	0.8

To compare the results of the different algorithms, we (arbitrarily) decided that both the strong network and disk perturbation should be considered anomalies, but not the weak injections. We defined a metric to maximize the correctly detected anomalies while minimizing false positives (FP), and false negatives (FN), by using the following formula:

$$\frac{(False\ Positives + False\ Negatives)}{Correctly\ Flagged\ Anomalies}$$

We call this metric the **badness** of the algorithm because a higher value corresponds to a less effective approach. Table III shows the number of false positives, false negatives, and the badness metric for all 8 algorithms, with the best scores highlighted.

The static approaches, despite high badness scores, had one clear advantage: they detected the latency injection. The heuristic approaches probably missed it because at this point in the run, the historical data contains many anomalies more severe than the transition to lower latency. Thus, an effective use of the static approach might be as a specialized method of discovering this type of fundamental change in behavior, instead of using it to discover transient anomalies. However, the two heuristic smoothed approaches had the best badness score overall and the fewest false positives, showing that algorithmic adaptations to avoid outliers give these approaches an advantage.

D. Five Hour Run Results

Our second set of experiments, the 5 hour run, examined these 8 algorithms in a system with randomized, and possibly overlapping, perturbations. We used the same Emulab configuration and repeatedly ran GridFTP transfers. Every two to three minutes, we would randomly choose to initiate a disk or network injection. Unlike the previous run, however, the length and degree of the disk and network injections were randomized to so that it was possible for one injected anomaly to overlap with the next. Partially overlapping weak injections

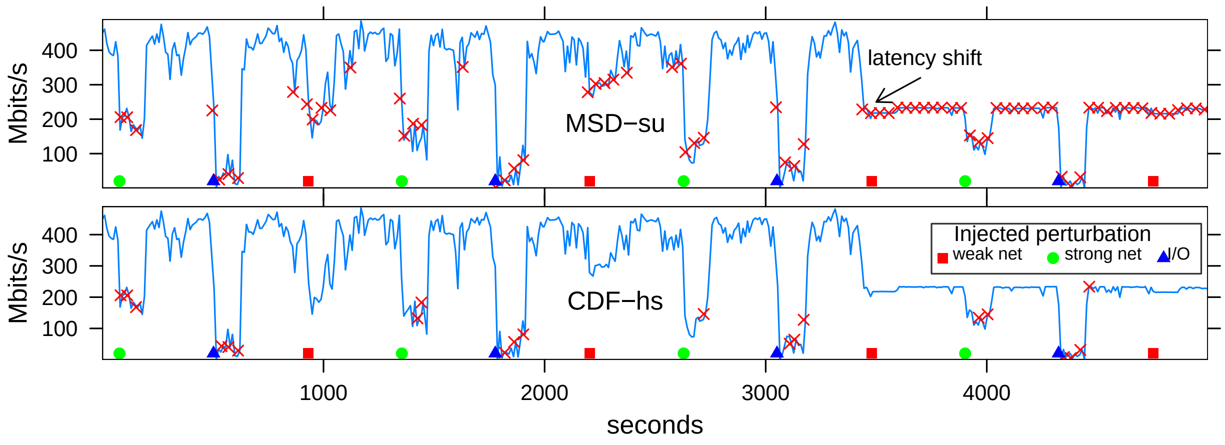

Fig. 6. GridFTP Performance with injected performance perturbations and detected anomalies.

TABLE IV

NUMBER OF I/O AND NETWORK INJECTIONS IN THE RUNS.

Type	Controlled Run	5 Hour Run
Disk I/O Injection	4	35
Weak Network Injection	4	25
Strong Network Injection	4	45
Overlapped Injections	0	8

TABLE V

EXPERIMENTAL ANOMALIES DETECTED.

Algorithm	File Transfer Anomalies
MSD-ss	153
MSD-su	136
MSD-hs	37
MSD-hu	18
CDF-ss	145
CDF-su	152
CDF-hs	64
CDF-hu	39

had a cumulative effect on performance that was similar to, but not the same as, a strong injection. In addition, we split the injections into two classes: more than three iperf streams or one disk injection stream was considered a strong network or I/O injection, everything else was considered a weak injection. The resulting number weak, strong, and overlapping injections over the course of the whole run is shown in the rightmost column of Table IV.

The number of anomalies reported by each of the 8 algorithms are shown in Table V. Due to the randomness of this experiment, there is no consistent basis for determining which perturbations are anomalies. Partially overlapping injections may or may not be anomalies according to the criteria used in the control data. Therefore, we simply compare the algorithms against each other.

All of the static approaches seemed over-sensitive, detecting approximately twice the number of non-overlapped anomalies injected into the system, and up to seven times as much as other approaches.

The higher variation in the data seemed to increase the false negative rate for the MSD–heuristic algorithms, with and without smoothing, with only 18 or 37 anomalies detected, or less than 1/3 the total number of disk and strong network injections (minus overlaps). The best approach in this environment, as it was in the controlled environment, appears to be CDF Heuristic with Smoothing (CDF-hs), with 64 anomalies detected. We believe that the non-parametric estimator outperforms the parametric one due to the non-normal nature of the data.

In summary, no single algorithm works best for all types of anomalies. This leads us to assert that robust anomaly detection of GridFTP should use multiple algorithms in parallel. Some higher level component can then customize the reported results for a given user or administrator. For example, if one want to know about baseline changes such as the latency shift shown in Figure 6, any of the static algorithms(CDF-ss, CDF-su, MSD-ss, MSD-su) should be selected, while if one wishes to detect the types of random anomalies in our 5-hour test, the CDF Heuristic with Smoothing (CDF-hs) method will work better.

E. *Experimental Summary*

We have seen that an appropriate algorithm can do a good job of appropriate anomaly detection. These algorithms are not only lightweight enough to be run online, they are fast enough to process batches of historical data. For example, for the results above the CDF algorithm takes approximately 10 seconds to process 5 hours worth of data and the MSD algorithm takes 5 seconds. This is about 1800 to 3600 faster than “real time”, providing plenty of headroom to scale up to a system with many input streams.

VI. RESULTS FOR A CPU-BOUND GRID JOB

We next evaluated whether the results for GridFTP carried over to a simple CPU-intensive job. We found that in fact the results were quite different, as described below.

A. *Experiment Configuration*

Using one Emulab pc3000 node, we ran a NetLogger-instrumented program that performs FFT computations using the popular FFTW [18] library. Once a minute, we ran a

TABLE VI
ANOMALIES DETECTED IN CPU-BOUND JOB DATA USING THE VARIOUS APPROACHES (OUT OF 53 POSSIBLE).

Algorithm	Correct	FP	FN	Badness
MSD-ss	**53**	**0**	**0**	**0**
MSD-su	53	1	0	0.019
MSD-hs	34	0	19	0.5588
MSD-hu	34	0	19	0.5588
CDF-ss	**53**	**0**	**0**	**0**
CDF-su	53	9	0	0.17
CDF-hs	32	0	21	0.656
CDF-hu	32	1	21	0.688

CPU injection. Over the one-hour test, 53 CPU injections were performed. Each of these injections was considered an anomaly, so we can again apply our *badness* metric to evaluate the results.

B. Results

The results of these tests are shown in Table VI, with the best scores highlighted. We can see that the static approach with smoothed input performs optimally. The heuristic approaches, which did well for the network data, suffer from many false negatives. This shows that in certain environments (like single machine CPU usage) the baseline can be effectively set and adaptation simply lowers the detector's sensitivity.

Although the CPU "job" presented here was trivial, we believe that its consistent performance profile is typical of a significant class of real applications. For these kinds of processes, a well-established baseline can lead to simple anomaly detection, e.g. a static baseline with mean and standard deviation. This further supports our belief that different algorithms are appropriate for different sorts of data.

VII. Previous and Related Work

In this section we review some related work on monitoring and data reduction, and anomaly prediction and identification.

A. Monitoring and Data Reduction

We have been involved in instrumenting and understanding application and Grid behavior for over 10 years. Our previous papers have described NetLogger performance [14], the need for dynamic instrumentation [19], and a NetLogger component to detect workflow anomalies [20]. This paper extends this work by introducing a NetLogger summarization component.

Sampling and clustering have been used by other monitoring tools to reduce the overhead of monitoring, mostly in the context of parallel computing. The Pablo monitoring system uses event throttling [6] to replace event tracing with counts by monitoring the observed event rate and comparing it to user-specified high and low water marks. Although throttling prevents generation of large data volumes, it sacrifices a consistent view of application behavior. TAU [7] includes an option to automatically eliminate monitoring of the smallest (thus, most perturbed) routines. However, this requires a training run, and unlike summarization or sampling removes all information about those routines. dynSIGMA [21] uses Dyninst's ability to dynamically activate instrumentation for sub-sampling of memory traces. The technique is interesting, but memory traces are too fine-grained for distributed computing, and current work doesn't extend to traces of arbitrary application events.

Data management and reduction have been widely studied in many areas, including medical data analysis [22], financial time series prediction [23], and biological data sampling [24]. Data reduction strategies chiefly rely on statistical techniques such as averaging, variance, covariance matrices, sampling, and principal component analysis. In previous work, we examined correlation elimination to diminish the volume of performance data [25] [26].

B. Prediction and Identification of Anomalies

Anomaly prediction and analysis is a core statistical pursuit, and its application to ongoing process control goes back at least to Shewhart's work on control charts in the 1930's [27]. Anomaly detection applied to intrusion detection and computer security has been an active area of research since it was originally proposed by Denning [28]. These approaches typically build a profile over the normal data and then check to see how well new data fits that profile [29].

Cottrell, et al. present a study comparing a number of techniques for detecting anomalies in network data [30]. While their techniques are effective, all are computationally quite expensive by comparison to our approach. The Network Weather Service efforts have explored forecasting poor network performance based on active network probing[31].

Application-level performance anomaly detection has been pursued by Allen et al. [32] to detect performance contract violations using a window average based method on the execution time of an application. Zhang et al. [33] show how to detect compliance with service-level objectives in a dynamic environment by managing an ensemble of Bayesian network models. Kelly [34] proposes how to use queuing theory observations to distinguish performance faults from overload. All of these are heavy weight approaches that are not generally applicable to near real-time detection.

Yang et al. [35] examine techniques for modeling cyclical performance variation. Mirgorodskiy and Miller [36] describe a method for dynamic instrumentation of executables that can be used to detect anomalies based on unusual control flow as opposed to unusual performance.

VIII. Conclusions

In this paper we present an infrastructure for troubleshooting performance problems in a large distributed system such as a Grid. We introduce a new component, an application library which performs application log summarization with minimal overhead, dramatically reducing the amount of log information while still preserving key performance information. We explore a variety of statistical methods to do anomaly detection, and conclude that no single method works for all types of anomalies. A Grid anomaly detection system should likely use multiple statistical methods in parallel, and combine and filter the results based the types of anomalies one is interested in learning about.

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. Additional support was provided by NSF NMI Award SCI-0438372.

REFERENCES

[1] "Open science grid (osg)," http://www.opensciencegrid.org/.

[2] I. Foster *et al.*, "Production grid: Principles and practice," *Proceedings of the IEEE International Symposium on High Performance Distributed Computing*, 2004.

[3] F. Berman, A. Chien, K. Cooper, J. Dongarra, I. Foster, D. Gannon, L. Johnsson, K. Kennedy, C. Kesselman, J. Mellor-Crummey, D. Reed, L. Torczon, and R. Wolski, "The GrADS Project: Software support for high-level Grid application development," *The International Journal of High Performance Computing Applications*, vol. 15, no. 4, pp. 327–344, 2001.

[4] G. Chun, H. Dail, H. Casanova, and A. Snavely, "Benchmark probes for grid assessment," in *Proceedings of the High-Performance Grid Computing Workshop*, 2004.

[5] W. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, and I. Foster, "The globus striped gridftp framework and server," in *In Proceeding of IEEE Supercomputing*, November 2005.

[6] D. A. Reed, R. A. Aydt, R. J. Noe, P. C. Roth, K. A. Shields, B. W. Schwartz, and L. F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," in *Proc. Scalable Parallel Libraries Conf.* IEEE Computer Society, 1993, pp. 104–113.

[7] J. C. B. Mohr, A. Malony, *Parallel Programming using C++*. M.I.T. Press, 1996, ch. TAU.

[8] A. V. Mirgorodskiy, N. Maruyama, and B. P. Miller, "Problem diagnosis in large-scale computing environments," in *Proceedings of IEEE SuperComputing '06*, November 2007.

[9] "Logging best practices guide." [Online]. Available: http://www.cedps.net/wiki/index.php/LoggingBestPractices/

[10] "Iso-8601: Data elements and interchange formats - information exchange - representation of dates and times," International Organization for Standardization, 1888. [Online]. Available: http://www.iso.ch/markete/8601.pdf

[11] D. Gunter, K. Jackson, D. Konerding, J. Lee, and B. Tierney, "Essential grid workflow monitoring elements," in *Proceedings of the International Conference on Grid Computing and Applications*, 2005.

[12] P. Leach, M. Mealling, and R. Salz, "A universally unique identifier (uuid) urn namespace," RFC4122, July 2005.

[13] "Syslog-ng," http://www.balabit.com/products/syslog-ng/.

[14] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer, "Dynamic monitoring of high-performance distributed applications," in *11th IEEE Symposium on High Performance Distributed Computing*, 2002.

[15] B. Tierney and D. Gunter, "Netlogger: A toolkit for distributed system performance tuning and debugging," LBNL, Tech. Rep. LBNL-51276, 2002.

[16] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," in *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*. Boston, MA: USENIX Association, Dec. 2002, pp. 255–270.

[17] J. S. H. G. E. P. Box, W. G. Hunter, *Statistics for Experimenters*. Wiley-Interscience, 1978.

[18] M. Frigo and S. G. Johnson, "The design and implementation of FFTW3," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005, special issue on "Program Generation, Optimization, and Platform Adaptation".

[19] D. Gunter, B. Tierney, C. E. Tull, and V. Virmani, "On-demand grid application tuning and debugging with the netlogger activation service," in *4th International Workshop on Grid Computing (Grid2003)*, 2003.

[20] D. Gunter and B. Tierney, "Scalable analysis of distributed workflow traces," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, 2005.

[21] J. Odom, L. DeRose, K. Ekanadham, J. Hollingsworth, and S. Sbaraglia, "Using dynamic tracing sampling to measure long running programs," *Proceedings of SuperComputing '05*, 2005.

[22] Y. Qu, B. Adam, M. Thornquist, J. Potter, M. Thompson, Y. Yasui, J. Davis, P. Schellhammer, L. Cazares, M. C. Jr., and Z. Feng, "Data reduction using a discrete wavelet transform in discriminant analysis of very high dimensionality data," in *Biometrics*, vol. 59, 2003.

[23] A. Lendasse, J. Lee, E. Bodt, V. Wertz, and M. Verleyen, "Input data reduction for the prediction of financial time series," in *Proceedings of the European Symposium on Artificial Neural Networks (ESANN'01)*, 2001.

[24] N. Yoccoz, J. Nichols, and T. Boulinier, "Monitoring of biological diversity in space and time," in *Trends in Ecology and Evolution*, vol. 16, 2001, pp. 446–453.

[25] M. Knop, J. Schopf, and P. Dinda, "Windows performance monitoring and data reduction using watchtower," in *Proceedings of Workshop on Self-Healing, Adaptive and self-MANaged Systems (SHAMAN)*, June 2002.

[26] L. Yang, J. Schopf, C. Dumitrescu, and I. Foster, "Statistical data reduction for efficient application performance monitoring," in *Proceedings of the Grid Workshop 2006*, October 2006.

[27] W. A. Shewhart, *Economic Control of Quality of Manufactured Product*. American Society for Quality, 1931.

[28] D. Denning, "An intrusion-detection model," *IEEE Transactions on Software Engineering*, pp. 222–232, 1987.

[29] A. Jones and R. Sielken, "Computer system intrusion detection: A survey," in *Tech Report, Computer Science Dept., University of Virginia*, 2000.

[30] R. Cottrell, C. Logg, M. Chhaparia, M. Grigoriev, F. Haro, F. Nazir, and M. Sandford, "Valuation of techniques to detect significant network performance problems using end-to-end active network measurements," in *SLAC-PUB-11653*, 2006.

[31] M. Swany and R. Wolski, "Multivariate resource performance forecasting in the network weather service," in *Proceedings of SC 2002*, November 2002.

[32] G. Allen, D. Angulo, I. Foster, and et al., "The cactus worm: Experiments with dynamic resource discovery and allocation in a grid environment," University of Chicago, Tech. Rep. Chicago TR-2001-28, 2001.

[33] S. Zhang, I. Cohen, M. Goldszmidt, and et al., "Ensembles of models for automated diagnosis of system performance problems," in *IEEE Conference on Dependable Systems and Networks (DSN)*, 2005.

[34] T. Kelly, "Detecting performance anomalies in global applications," in *Second USENIX Workshop on Real, Large Distributed Systems (WORLDS 2005)*, 2005.

[35] L. Yang, "Anomaly management in grid environments," in *PhD Thesis,, University of Chicago, Computer Science Department*, 2007.

[36] A. V. Mirgorodskiy and B. P. Miller, "Diagnosing distributed systems with self-propelled instrumentation," in *University of Wisconson Technical Report*, 2007.