# DCAMP: DISTRIBUTED COMMON API FOR MEASURING PERFORMANCE

A Thesis

Presented to

the Faculty of California Polytechnic State University

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Alexander Paul Sideropoulos

June 2009

COMMITTEE MEMBERSHIP

| | |
|---|---|
| TITLE: | dCAMP: Distributed Common API for Measuring Performance |
| AUTHOR: | Alexander Paul Sideropoulos |
| DATE SUBMITTED: | June 2009 |
| COMMITTEE CHAIR: | Dr. Michael Haungs |
| COMMITTEE MEMBER: | Dr. Gene Fisher |
| COMMITTEE MEMBER: | Dr. David Janzen |

**Abstract**

dCAMP: Distributed Common API for Measuring Performance

Alexander Paul Sideropoulos

Although the nearing end of Moore's Law has been predicted numerous times in the past [10], it will eventually come to pass. In forethought of this, many modern computing systems to become increasingly complex, distributed, and parallel. As software is developed on and for these complex systems, a common API is necessary for gathering vital performance related metrics while remaining transparent to the user, both in terms of system impact and ease of use. Several distributed performance monitoring and testing systems have been proposed and implemented by both research and commercial institutions. However, most of these systems do not meet several fundamental criterion for a truly useful distributed performance monitoring system: 1) variable data delivery models, 2) security, 3) scalability, 4) transparency, 5) completeness, 6) validity, and 7) portability [14]. This work presents the Distributed Common API for Measuring Performance, *dCAMP* , as a solution to meet each of these criterion.

# Acknowledgements

Thank you...

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

As the Internet has become more pervasive in today's business economy, there has been a natural trend of distributing large, complex systems across multiple components locally and throughout the world. These systems are not always homogeneous with respect to hardware architecture or even operating system, and development of these system can prove to be quite difficult even with the best tools available. In order to effectively build these systems, software engineers must be able to test their system for performance defects as well as bottlenecks. Additionally, distributed systems must respond to changes in availability and work load on its individual nodes.

Distributed performance testing frameworks supply software practitioners and system administrators with tools to evaluate the performance of a system from both black box and white box perspectives by publishing interfaces for instrumenting, collecting, analyzing, and visualizing performance data across the distributed system and distributed applications. Distributed performance monitoring frameworks, often considered part of the testing framework, provide a black box interface into monitoring a distributed system or application and usually

includes mechanisms for triggering actions based on performance events. For the purpose of this work, I introduce the term distributed performance framework to collectively refer to both distributed performance testing and distributed performance monitoring frameworks.

## 1.1 Distributed Performance Framework Criterion

In order for practitioners and researchers alike to effectively use a distributed performance framework, it is necessary to have a set criteria for evaluation. Presented here is an extended criterion of the general requirements presented by [14] for grid systems. Data Delivery Models and Security have been taken directly from their work. Scalability has been modified to only consider good performance as its goal while Low Intrusiveness has been turned into Transparency. Extensibility has been removed from the list, and Completeness and Validity have been added. This work provides an alternate definition for Portability.

### 1.1.1 Data Delivery Models

Monitoring information includes fairly static (e.g., software and hardware configuration of a given node) and dynamic events (e.g., current processor load, memory), which suggests the use of different measurement policies (e.g., periodic or on demand). In addition, consumer patterns may vary from sparse interactions to long lived subscriptions for receiving a constant stream of events. In this regard, the monitoring system must support both pull and push data delivery models. [14]

### 1.1.2 Security

Certain scenarios may require a monitoring service to support security services such as access control, single or mutual authentication of parties, and secure transport of monitoring information. [14]

### 1.1.3 Scalability

Monitoring systems have to cope efficiently with a growing number of resources, events and users. This scalability can be achieved as a result of good performance which guarantees that a monitoring system will achieve the needed throughput within an acceptable response time in a variety of load scenarios. [14]

### 1.1.4 Transparency

Transparency refers to the lack of impact a distributed performance framework makes on the system being monitored. As [14] states, it is "typically measured as a function of host (processor, memory, I/O) and network load (bandwidth) generated by the collection, processing and distribution of events." If a framework lacks transparency it will fail to allow the underlying distributed system to perform well and will produce inaccurate performance measurements, thereby reducing its Scalability and destroying its Validity.

### 1.1.5 Completeness

The Completeness of a distributed performance framework refers to the exhaustiveness to which it gathers performance metrics. At a minimum, a framework must provide interfaces for measuring and aggregating performance data

about a system's processor, memory, disk, and network usage. Several distributed performance frameworks provide further detailed performance metrics about the given distributed system being monitored, but this is usually at the cost of Portability.

### 1.1.6 Validity

A distributed performance framework is only as good as the data is produces; if the sensors or gathering techniques are inaccurate, then the data is inaccurate and useless. Validity of a framework is achieved when the authors of a framework provide formal verification of its accuracy.

### 1.1.7 Portability

A framework's ability to run on a completely heterogeneous distributed system without special considerations by the practitioner is what I define as Portability. More specifically, a portable framework has a unified API regardless of the system architecture, does not restrict itself to applications written in specific programming languages, and does not require practitioners to manually instrument their application code. This black box characteristic is vital for a viable distributed performance framework's effectiveness as it allows practitioners to focus on the performance data and not on a myriad of APIs for various architectures or languages.

# Chapter 2

# *dCAMP*

The Distributed Common API for Measuring Performance (*dCAMP* ) is a distributed performance framework built on top of Mark Gabel and Michael Haungs' 2007 research on *CAMP: a common API for measuring performance* [5]. The fundamental functionality of *CAMP* is providing an accurate and "consistent method for retrieving system performance data from multiple platforms." *dCAMP* takes advantage of this functionality and the authors' work done in validating *CAMP* 's accuracy and adds the following core feature sets:

- Stateful Performance API
- Distributed Performance Data Aggregation
- Performance Filters and Triggers
- Simplistic Fault Tolerance

## 2.1 Terminology

Knowing the following terminology will make it easier to understand and discuss the *dCAMP* project, its main goals, its usage, its components, and how it works.

**Distributed Performance Testing Framework (DPTF),**

**Distributed Performance Monitoring Framework (DPMF),**

**Distributed Performance Framework (DPF):** An DPTF or DPMF (collectively termed DPF) is a framework which allows its users to evaluate the performance of a system from both black box and white box perspectives by publishing interfaces for instrumenting, collecting, analyzing, and visualizing performance data across the distributed system and distributed applications. Typically, the framework provides a black box interface into monitoring a distributed system or application and includes mechanisms for triggering actions based on performance events. The *dCAMP* project is designed to be an DPF.

**Performance Metric:** Performance metrics are any data about a given node relating to its throughput, capacity, utilization, or latency. In *dCAMP* , these are grouped into four different sets of performance metrics (global, network, disk, and per-process) and a fifth set of inquiry metrics. They are described fully below.

**Metric Collection,**

**Metric Aggregation,**

**Metric Calculation:** Metric collection (or aggregation or calculation) is the process of combining metrics from multiple nodes into a single metric. Performance metrics, while useful at an individual system granularity, can be

rather limited in value for a DPF where the goal is measurement of the distributed system as a whole. Metric collection provides a coarser granularity for the performance metrics. Collection could calculate a sum, average, percent, or any other mathematically relevant operation across multiple nodes in the system.

**Filter,**

**Throttle,**

**Threshold:** Filtering (or throttling or thresholding) provides a mechanism for reducing the amount of data sent between nodes of the system. Filtering allows a user to specify when or at what point to report metrics from one level to its parent. For example, a filter might be set to only report average CPU utilization that is over seventy-five percent.

***dCAMP* Node,**

***dCAMP* Process:** A single, independently running instance of *dCAMP* in the distributed system is called a *dCAMP* node or process. More than one node may exist on a single computer. A node consists of the Node role and zero or more other *dCAMP* roles.

***dCAMP* Role:** Roles are a way of logically grouping functions within the *dCAMP* system, from performance metric gathering to *dCAMP* system management. A description of all the *dCAMP* roles can be found below.

***dCAMP* Service:** Services in the *dCAMP* system are implementations of one or more *dCAMP* roles. There does NOT exist a one-to-one relationship between services and roles; the *dCAMP* service-to-role mapping can be seen below. Each services is implemented in *dCAMP* as a single Python class.

***dCAMP* Hierarchy:** The *dCAMP* system is organized in a hierarchical pat-

tern with respect to data movement and system control functionality. The hierarchy can be thought of as a tree structure, with leaf nodes being at the top of the hierarchy and a single root node at the bottom. Metric data moves down the hierarchy from leaves to the root; configuration data and control commands move up from the root to the leaves.

***dCAMP* Level:** Levels are a way of organizing the *dCAMP* hierarchy horizontally. Levels are defined by their distance from the root node. For example, level one is one node away from the root node, or said another way, the first level is directly "connected" to the root node. The second level is two nodes away from the root node, or any node in the second level is connected to the root node by another node (in the first level). This necessarily means the root is in level zero (all by itself).

**Parent Node:** Nodes are called parent nodes if there is at least one node connected to it from a level of higher ordinal value. For example, a node in level one with at least one node connected to it from level two is considered a parent node. The root node is inherently a parent node.

**Child Node:** A node is called a child node if it is connected to another node in a level of lower ordinal value. For example, a node in level one is connected to the root node (in level zero), so it is called a child node. The root node is the only node in the *dCAMP* system which is NOT a child node.

**Configuration:** The *dCAMP* configuration specifies everything about the system, including root node, hierarchy levels, metrics, sampling frequency, reporting frequency, filtering, communication details, etc. The configuration is set at the root node and then distributed to the rest of the *dCAMP* system. Configuration details can be found below.

**Metric Sampling:** Metric sampling is the process of measuring metrics on a

given node.

**Metric Reporting:** Metric reporting is the process of sending sampled metrics to a parent node.

## 2.2 *dCAMP* Metrics

Metrics marked with "(*dCAMP* )" are extensions added by the *dCAMP* project to the basic CAMP metrics. These provide a performance view of multiple nodes in the distributed network and are collected by the Aggregate service rather than the CAMP service.

### 2.2.1 Global Metrics

Global metrics measure overall CPU, process, thread, and memory usage of the system.

- Node CPU usage
- Node free physical memory (KB)
- Aggregate average CPU usage (*dCAMP* )
- Aggregate free physical memory (percentage) (*dCAMP* )

### 2.2.2 Network I/O Metrics

Network metrics measure utilization of a given network interface on the system.

- Total bytes sent on the given interface
- Total packets sent on the given interface

- Total bytes received on the given interface
- Total packets received on the given interface
- Aggregate bytes sent (*dCAMP* )
- Aggregate packets sent (*dCAMP* )
- Aggregate bytes received (*dCAMP* )
- Aggregate packets received (*dCAMP* )

### 2.2.3 Disk I/O Metrics

Disk I/O metrics measure throughput of a given disk or partition on the system.

- Number of read operations on the given disk
- Number of write operations on the given disk
- Number of read operations on the given partition
- Number of write operations on the given partition
- Aggregate number of read operations (*dCAMP* )
- Aggregate number of write operations (*dCAMP* )

### 2.2.4 Per-process Metrics

Per-process metrics measure CPU, memory, and thread usage for a single process on the system.

- Number of major and minor page faults
- Process CPU utilization
- Process user mode CPU utilization
- Process privileged mode CPU utilization

- Size of the process' working set in KB
- Size of the used virtual address space in KB
- Number of threads contained in the process

### 2.2.5 Inquiry Metrics

Inquiry metrics provide a mechanism for enumerating various properties of the system.

- Enumeration of the available disk partitions
- Enumeration of the available physical disks
- Enumeration of the valid inputs to the network functions
- Number of CPUs in the system
- "process identifier" for the given pid
- "process identifier" for each running process launched from an executable of the given name

# Chapter 3

# Design & Implementation

**General design:** semi-centralized, hierarchical peer-to-peer system utilizing the pipe-flow architecture pattern in which leaf (sensor) nodes of the hierarchy collect data, filter out extraneous data, and send it up the pipe to an aggregate node which subsequently filters out more data and sends it up to another aggregate or root node.

## 3.1 *dCAMP* Roles and Services

### 3.1.1 Roles

- **Node**—basic dCAMP functionality; provides inter-node communication, heartbeat monitoring, and failure recovery.
- **Sensor**—local performance metric gathering; essentially the dCAMP layer on top of the OS and hardware performance APIs.
- **Filter**—performance metric filtering; provides throttling and thresholding of metrics.

- **Collector**—performance metric aggregation; provides collection of and calculation on metrics from multiple sensors and/or collectors.
- **Management**—primary entry-point for end-user control of dCAMP system; provides hierarchy management and configuration distribution.

### 3.1.2 Service-to-Role Mapping

The following is a table of services which can be "published" by a dCAMP node and the roles which they implement. The base service is running on every node in the dCAMP system. The root service is essentially a special aggregate service which provides the additional management role.

| **Service** | **Role(s)** |
|---|---|
| Root Service | Management, Collector, Filter |
| Aggregate Service | Collector, Filter |
| Metrics Service | Sensor, Filter |
| Base Service | Node |
| CAMP Service | (none) |

### 3.1.3 Failure Rules

- if sensor loses its aggregate, communicate with root to be assigned to a new aggregate
- if sensor loses its root, it assumes the root role
- to reconcile multiple root roles in one system, one root notifies other roots of its superiority; new root assumes responsibility for restructuring hierarchy and coordinating data synchronization process

### 3.1.4 Limitations

- uniform configuration: The configuration of the entire system is uniform, i.e., all branches of the system will be collecting the same data at each level in the hierarchy—it is not possible to have different data being collected by different branches.
- network failure: *dCAMP* does not support any fault tolerance for network failures; *dCAMP* only attempts to recover from node failures. It is assumed that if (part of) the network goes down, the lack of data from that subnet will suffice
- time accuracy: The system time among multiple nodes in the system may vary significantly; *dCAMP* is not meant to be a high-resolution system with respect to the order of performance data occurrences. It is assumed that the ordering of performance events in the system is insignificant and timestamps associated with performance data are rough estimates.

### 3.1.5 Execution Steps

These steps are for the current, prototype version of *dCAMP* . *Setup: All nodes are running as base nodes (listening for commands via CLI and network port)*

1. User initializes root node via CLI (providing configuration)
2. Root node registers each listening node (configuring as metric nodes)
3. Metrics are sampled, reported, and logged (repeating indefinitely)
4. Metric nodes sample metrics and report metrics back to root node
5. Root node logs received metrics to disk
6. User shuts down root node via CLI

7. Root node unregisters each listening node (reverting to base nodes)
8. Root node terminates (reverting to base node)

# Chapter 4

# Analysis

To verify that *dCAMP* meets the distributed performance framework criterion outlined in 1, several experiments were run on a test installation of *dCAMP* in a test environment. The goal of these experiments was two fold: verify *dCAMP* 's functionality as well as determine the thresholds for several key configuration parameters. For *dCAMP* to scale, it is important for the number of child nodes per parent to be limited to a reasonable number. These experiments help to define "reasonable" for various scenarios, environments, and performance monitoring requirements.

*list out some experiments, test environment characteristics, etc.*

# Chapter 5

# Related Work

*THIS SECTION IS FROM MY 508 PAPER*
*AND NEEDS TO BE CLEANED UP*

Being distributed, a framework must collect data from a large number of nodes and aggregate the data to one node or client. Implementations have been built using centralized, hierarchical, peer-to-peer and any number of other architectures. There are three types of metric gathering techniques: (1) hardware counters and sensors use specialized hardware to gather highly accurate metrics and are highly dependent on the underlying hardware architecture, (2) software sensors use modern operating system interfaces to acquire moderately accurate performance metrics in an architecture-independent interface, and (3) hybrid approaches use a combination of hardware and software sensors to attain a balance between the two.

There are a number of distributed performance frameworks being actively researched and developed, both academically and commercially. This work defines a set of criteria for evaluating distributed performance frameworks to measure the

usefulness and viability of the framework. In the next section I list similar survey work, showing the novelty of this paper. Section 3 presents the evaluation criteria, section 4 provides the evaluation results of several distributed performance frameworks, and section 5 concludes.

## 5.1 Related Work

Several papers have been published dealing with this topic and which try to give an overview of the current state of distributed performance frameworks. This work, while related, differs in two main aspects: (1) the evaluation criteria presented here is more complete than previous works' requirements and (2) this work looks specifically at distributed performance frameworks rather than distributed system or grid tools and frameworks.

Zoraja et al. include a section on Monitoring Issues in their work on OMIS/OCM, CORAL, and MIMO [15] focusing on design and implementation issues for monitoring systems. These monitoring issues share some critical points with the criteria presented here, but the work lacks an analysis of other distributed performance frameworks and only includes limited analysis of the presented frameworks against the listed monitoring issues. [8] provides a short survey of related monitoring systems in comparison to the ZM4/SIMPLE monitoring environment. While this survey does look at a number of important framework characteristics, it is restricted to hybrid distributed performance frameworks.

Allan and Ashworth published an exhaustive survey of distributed computing tools [2] but did not present any particular focus on distributed performance frameworks nor did they include a set of criteria for evaluating frameworks. Zanikolas et al. present an in-depth taxonomy of grid monitoring systems [14],

including a set of general requirements for monitoring systems. Their work does much for defining a good set of criteria for evaluating and analyzing distributed performance frameworks, but has a much broader scope than this work. This work expands on their monitoring system requirements to account for the completeness and validity of distributed performance frameworks.

## 5.2 Analysis Results

The distributed performance frameworks analyzed in this section were chosen based on their categorization in the [14] taxonomy; only level 2 frameworks are included. Level 2 frameworks are defined as having at least one type of republisher in addition to producers; these frameworks usually distribute functionality across multiple hosts. [14] A limited analysis is conducted by reviewing the available literature, and further analysis (i.e., verifying scalability, transparency, and validity) is left as future work.

### 5.2.1 NetLogger

Work done by Brian Tierney and Dan Gunter [12] [6] presents the Network Application Logger Toolkit (NetLogger). This framework can be used to monitor the performance of distributed systems at a very detailed level. With a new logging format and activation service [7] the authors improved upon their previous work and increased the toolkit's scalability and data delivery models. NetLogger is being actively developed and is one of the more well known distributed performance frameworks. The toolkit is composed of four parts: an API and library for instrumenting a given application, a set of tools for collecting and sorting logs,

performance sensors, and a visualization user interface for the log files.

Each part assumes the system clocks of the individual nodes are accurate and synchronized (the authors mention the use of NTP to achieve a required clock synchronization of one millisecond). The instrumentation of code allows NetLogger to gather more detailed data from an application-to-application communication path, such as traces of network packets through a call hierarchy. The instrumentation also allows the activation service to update the monitoring of parts of the system dynamically as consumers subscribe to various events and metrics.

Their research has shown NetLogger to be highly scalable, complete, and transparent as well as valid. The activation service provides a push data delivery and can utilize the security mechanisms part of current web services in order to authenticate requests for performance data. NetLogger is currently implemented for C, C++, Java, Perl, and Python applications. Because the framework lacks black box characteristics, its portability is greatly reduced.

### 5.2.2 JAMM

Java Agents for Monitoring and Management (JAMM) [11] is the fruit of work by the authors of NetLogger to build a monitoring system with managed sensors.

The JAMM system consists of six components: sensors, sensor managers, event gateways, directory service, event consumers, and event archives. There is a sensor manager on each host, with the sensors acting as producers for the gateways which they publish the data to. The gateways can then filter and aggregate the incoming data according to consumer queries. The directory service is used

to publish the location of the sensors and gateways, allowing for dynamic discovery of active sensors by the consumers. The event archive is used for historical analysis purposes.

JAMM explicitly uses a pull data delivery model where data is only sent when requested by a consumer. The overall architecture is generally distributed with the directory service being centralized. JAMM, being heavily based off of NetLogger, inherits the validity, completeness, security, and transparency of NetLogger along with its lack of portability. JAMM does, however, prove itself in terms of scalability with it's own architecture.

### 5.2.3 Hawkeye

Hawkeye [1] is a monitoring and management tool for distributed systems which makes use of technology previously researched and developed as part of the Condor project [9]. Condor provides mechanisms for collecting information about large distributed computer systems. Hawkeye is being readily developed and is freely available for download on Linux and Solaris.

Hawkeye uses a general push delivery model by configuring Condor to execute programs, or modules, at given time intervals, collect performance data, and send it to the central manager. These modules are configurable such that the "period' of module execution can be set to a given time frame in seconds, minutes, or hours or the module can be executed in "continuous" mode where the module's execution never ends. The available modules for monitoring a Condor pool include: disk space, memory used, network errors, open files, CPU monitoring, system load, users, Condor Node, Condor Pool, and Grid Probe. Custom modules can also be developed and installed for monitoring of arbitrary resources

and metrics. Data can be accessed from the central manager via an API, CLI, or GUI.

While no experiments have been run, the generally centralized manager reduces the Hawkeye framework's scalability, and its transparency is unknown. The frameworks module based producer architecture gives it an infinite completeness, but being only available on Linux and Solaris makes the framework less portable. Lastly, the ability to run jobs securely on target machines has been left as future work by the authors.

### 5.2.4 SCALEA-G

Truong and Fahringer present SCALEA-G [13], an unified monitoring and performance analysis system for distributed systems. It is based on the Open Grid Service Architecture [4] and allows for a number of services to monitor both grid resources and grid application. SCALEA-G uses dynamic instrumentation to profile and trace Java and C/C++ applications in both push and pull data delivery models, making the framework both scalable and portable.

The SCALEA-G framework is composed of several services: directory service, archival service, sensor manager service, instrumentation service, client service, and user portal. These services provide the following functionality respectively: publishing and searching of producers and consumers, storage of performance results, management of sensors, dynamic instrumentation of source code, administering clients and analyzing data, and on-line monitoring and performance analysis.

The framework makes use of secure sockets to achieve secure communications and achieves high completeness via code instrumentation. Unfortunately, the

authors do not provide any report on SCALEA-G's validity or transparency.

### 5.2.5 IMPuLSE

Integrated Monitoring and Profiling for Large Scale Environments [3] was designed to address "operating system-induced performance anomalies" and provide "accurate, low-overhead, whole-system monitoring." The authors have chosen to develop a message-centric approach which associates data with messages rather than hosts and a system-wide statistical sampling to increases the framework's scalability.

The IMPuLSE framework is still in the design stage, and therefore lacks any implementation data outside of their new message-centric design pattern which shows promising results. Unfortunately, this leaves the framework with unknown transparency, security, completeness, portability, and validity.

## 5.3 Conclusion

There are a number of high quality and effective distributed performance frameworks being actively researched and developed, but with some frameworks having more research than others, there is a natural disparity of information about each framework. While the frameworks vary in distributed architecture and features, they all fulfill the minimum requirements of performance frameworks. The frameworks analyzed in this work are mainly software based sensor frameworks. This was chosen due to the inherent portability advantage of software sensors over hardware or hybrid sensors.

Many authors have failed to address their framework's validity, transparency,

and scalability explicitly, thinking the framework's architecture speaks for itself or blindly assuming it is accurate and introduces negligible load on the measured system. I leave it as future work to conduct formal experiments to test validity, transparency, and scalability of the distributed performance frameworks analyzed here.

# Chapter 6

# Future Work

# Chapter 7

# Conclusions

# Bibliography


[1] Hawkeye http://www.cs.wisc.edu/condor/hawkeye/.

[2] R. Allan and M. Ashworth. A survey of distributed computing, computational grid, meta-computing and network information tools, 2001.

[3] P. G. Bridges and A. B. MacCabe. Impulse: integrated monitoring and profiling for large-scale environments. In *LCR '04: Proceedings of the 7th workshop on Workshop on languages, compilers, and run-time support for scalable systems*, pages 1–5, New York, NY, USA, 2004. ACM.

[4] I. Foster, C. Kesselman, J. M. Nick, and S. Tuecke. Grid services for distributed system integration. *Computer*, 35(6):37–46, 2002.

[5] M. Gabel and M. Haungs. Camp: a common api for measuring performance. In *LISA'07: Proceedings of the 21st conference on Large Installation System Administration Conference*, pages 1–14, Berkeley, CA, USA, 2007. USENIX Association.

[6] D. Gunter, B. Tierney, B. Crowley, M. Holding, and J. Lee. Netlogger: A toolkit for distributed system performance analysis. *International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, 0:267, 2000.

[7] D. Gunter, B. Tierney, K. Jackson, J. Lee, and M. Stoufer. Dynamic monitoring of high-performance distributed applications. In *Applications, Proceedings of the 11th IEEE Symposium on High Performance Distributed Computing*, pages 163–170, 2002.

[8] R. Hofmann, R. Klar, B. Mohr, A. Quick, and M. Siegle. Distributed performance monitoring: Methods, tools and applications. *IEEE Transactions on Parallel and Distributed Systems*, 5:585–597, 1994.

[9] M. Litzkow, M. Livny, and M. Mutka. Condor-a hunter of idle workstations. *Distributed Computing Systems, 1988., 8th International Conference on*, pages 104–111, Jun 1988.

[10] R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, Jun 1997.

[11] B. Tierney, B. Crowley, D. Gunter, J. Lee, and M. Thompson. A monitoring sensor management system for grid environments. In *Proceedings of the IEEE High Performance Distributed Computing Conference (HPDC-9)*, pages 97–104, 2000.

[12] B. Tierney, W. Johnston, B. Crowley, G. Hoo, C. Brooks, and D. Gunter. The netlogger methodology for high performance distributed systems performance analysis. In *In Proc. 7th IEEE Symp. on High Performance Distributed Computing*, pages 260–267, 1998.

[13] H.-L. Truongc and T. Fahringer. Scalea-g: A unified monitoring and performance analysis system for the grid. *Sci. Program.*, 12(4):225–237, 2004.

[14] S. Zanikolas and R. Sakellariou. A taxonomy of grid monitoring systems. *Future Gener. Comput. Syst.*, 21(1):163–188, 2005.

[15] I. Zoraja, G. U. Rackl, and T. Ludwig. Towards monitoring in parallel and distributed environments. In *University of Split*, pages 133–141, 1999.