QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems

Sandip Agarwala*, Yuan Chen†, Dejan Milojicic† and Karsten Schwan*
*College of Computing
Georgia Institute of Technology
Atlanta, GA 30332
{sandip, schwan}@cc.gatech.edu
† Hewlett Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304
{yuan.chen, dejan.milojicic}@hp.com

***Abstract*— The scale, reliability, and cost requirements of enterprise data centers require automation of center management. Examples include provisioning, scheduling, capacity planning, logging and auditing. A key component of such automation functions is online monitoring. In contrast to monitoring systems designed for human users, a particular concern for online enterprise monitoring is Quality of Service (QoS). Since breaking service level agreements (SLAs) has direct financial and legal implications, enterprise monitoring must be conducted so as to maintain SLAs. This includes the ability to differentiate the QoS of monitoring itself for different classes of users or more generally, for software components subject to different SLAs. Thus, without embedding notions of QoS into the monitoring systems used in next generation data centers, it will not be possible to accomplish the desired automation of their operation.**

This paper both demonstrates the importance of QoS in monitoring, and it presents a QoS-capable monitoring system, termed QMON. QMON supports utility-aware monitoring while also able to differentiate between different classes of monitoring, corresponding to classes of SLAs. The implementation of QMON offers high levels of predictability for service delivery (i.e., predictable performance), and it is dynamically configurable to deal with changes in enterprise needs or variations in services and applications. We demonstrate the importance of QoS in monitoring and the QoS capabilities of QMON in a series of case studies and experiments, using a multi-tier web service benchmark.

I. Introduction

Modern enterprises are characterized by growing dynamism, heterogeneity, complexity, and scale. It is widely accepted that next generation enterprises will increasingly require automated management [16], [31]. Management tools include commercial products like Tivoli [27] and HP's Openview [22] and academic efforts such as those focused on scientific workflows [2], [7]. Control systems and services may concern specific subsystems (e.g., database backend [17], [12] or carry out general tasks such as job scheduling [15], resource allocation [25], and problem diagnosis [11]. Regardless of their specific tasks and purposes, all such tools and services base their decisions on the online monitoring of the services, systems, or IT infrastructure they manage. Further, each of them will have different monitoring requirements in terms of the types of monitoring data to be acquired, the lifespan of that data, its timeliness or staleness properties, its precision or granularity, or even the jitter experienced during data acquisition (e.g., when controlling iterative or multimedia systems).

When applications must meet certain Service Level Objectives (SLOs), then the monitoring actions required for online application management must themselves meet certain levels of Quality of Service (QoS). For example, a front-end web request scheduler making online scheduling and dispatching decisions in a multi-tier web service [6] requires real-time data about the utilization levels experienced by backend servers, with timeliness requirements that are typically in the range of seconds. In comparison, a performance manager tracking an enterprise application's behavior by displaying data in a GUI will require levels of timeliness varying from seconds to minutes, depending upon the importance and SLO of the application or application component being monitored. A resource allocation system managing a large pool of compute servers may be subject to hourly shifts in usage due to west/east coast time differences, for example. Finally, a long term problem diagnosis program may use historical performance data to do root cause analysis. The deadlines associated with the monitoring data it requires may be in terms of days or weeks.

Monitoring requirements not only differ across applications or application components, but they also change over time. For example, the performance monitor of a 'silver' service requires monitoring data in the range of seconds, but when this service is upgraded to 'golden', the QoS demands imposed on its monitoring data become more stringent time-wise and in the level of detail required. Another example is an online job scheduler. It may not need fine-grained monitoring information under normal operating conditions, but when the workload rate increases, the scheduler will require more detailed and uptodate information about current resource availability and consumption.

While QoS in monitoring is a necessity for online management, most current commercial enterprise monitoring and tools [24], [27] are targeted at relatively static or slowly changing environments and therefore, do not provide the necessary

mechanisms to support adaptive monitoring and dynamic QoS guarantees. Finally, the management of monitoring systems in prior work has often relied on manual methods. Examples include the extensive monitoring facilities constructed for computer networks, enabling rich methods for manually changing the data capture, collection, and analysis methods applied to networks [32]. In comparison, automated techniques for changing the way in which systems are monitored have often focused on specific domains or applications (e.g., real-time systems [10], [28]), or they provide limited methods for configuring or self-configuring monitoring actions [1].

This paper proposes an approach to adaptive program monitoring with dynamic QoS guarantees. The QMON monitoring infrastructure described and evaluated in our research builds on a publish/subscribe monitoring paradigm, to provide monitoring capabilities to both local and remote management tools or services. Single or multiple users can dynamically subscribe to (or unsubscribe from) monitoring channels, thereby providing a rich infrastructure for both local and remote enterprise monitoring. More importantly, a set of programmable APIs enable the dynamic configuration of QMON channels: to change data collection, aggregation, correlation, and schedule. Specifically, QMON monitoring channels can be configured at runtime to change data collection parameters (i.e., what data to collect), the frequency of such data collection, the choices made for local data aggregation (i.e., the precision of monitoring data delivered to remote managers), the ways in which data is delivered to remote managers (i.e., monitoring granularity), and other QoS metrics associated with online monitoring. Furthermore, by associating QoS attributes with individual monitoring channels, different channels can have different attributes, thereby providing to higher level managers the ways to differentiate QoS levels for different monitoring tasks. Finally, by enabling dynamic channel creation, new QoS requirements and methods for attaining them can be deployed whenever or wherever needed in the enterprise.

While QMON may be used and configured explicitly, its interfaces are designed for interaction with higher level policy engines. These engines use enterprise-level policies to automatically manage channel creation, the assignments of users to channels, and similar higher level tasks. Further, the manner in which online monitoring is carried out is driven by current application needs, or, stated more precisely, monitoring is performed so as to continuously maximize the utility attained by the application being monitored and managed.

II. Monitoring and QoS

Automated management in next generation enterprise systems must consider multiple facts, including that system services must meet well-defined SLOs while also adapting to application and workload changes. Further, some workloads may be more important than others, with different associated business values or utilities [31]. Figure 1 shows the utility achieved from an application server of the Airline Reservation System run by one of our industrial partners. In this system, utility($\cup$) is defined as follows:

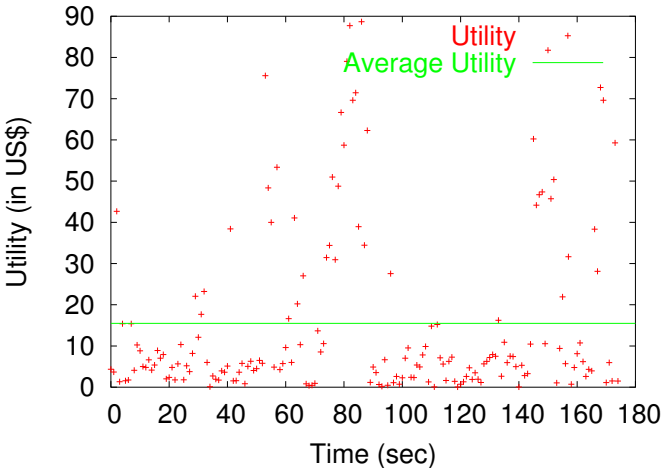

Fig. 1. Utility obtained from a server of our partner's Airline Reservation System

$$\cup \quad \propto \quad \frac{Number\ of\ results\ returned\ for\ a\ query}{Response\ Time}$$

Here, the utility of the result is dependent not only on raw performance (i.e., response time), but also on the quality of the data returned. The overall utility averaged over a 3 minute time window turns out to be "acceptable" ($15.49). But if the result is analyzed more carefully, we find that 75% of the transactions generated utility below the average. The average utility in the first minute of the time window is just $10.36 (66% of the total time window average!) From a business point of view, this is clearly unacceptable, as it may cause a business class passenger to experience low performance, which means less revenue for the company. In fact, previous research has shown that the average user's tolerance for delay in an e-commerce transaction is less than 11 seconds [5]. The study showed that an user has more tolerance towards low latency interactive response than high latency detailed response.

The above discussion demonstrates that enterprise systems like these must be designed and monitored in a way that allows 'micro' resource management, to ensure that achieved utility is high even on short time scales. This presents challenges to online management and monitoring, in part due to the overheads both impose on underlying systems. Factors on which overheads depend include the frequency with which a system is monitored, the metrics being collected, the number of network messages generated by the monitoring system, etc. Aggregation and correlation of monitoring data incur additional processing, storage and network bandwidth costs.

A concrete example of the overheads associated with monitoring appears in Table I, which shows the number of monitoring messages generated by the OVTA monitoring system in a sample installation of RUBiS. HP OpenView Transaction Analyzer (OVTA) [24] is a widely used commercial product based on the ARM specification [3], able to measure and analyze end-to-end transaction responses for WEB, J2EE, and COM applications. RUBiS is an open source multi-tier online auction benchmark from Rice University [8]. It implements the

TABLE I
OVTA MONITORING OVERHEAD

Levels	Monitoring records generated /min		CPU Utilization (in %)	
	Light workload	Heavy workload	Light workload	Heavy workload
No monitoring	0	0	17.70	28.88
Level 1	30	27	26.54	40.37
Level 2	443	735	28.07	45.88
Level 3	592	951	27.01	42.56
Level 4	4312	7347	27.46	43.32

core functionalities of an auction site like selling, browsing and bidding. Table I shows different levels of monitoring for two different types of workload. Each level produces a different number of end-to-end transactional records. In this experiment, these levels are created by manually configuring OVTA and selectively enabling its monitoring features. Increases indicate improvements in the levels of detail collected from the RUBiS system. As indicated in the table, these numbers can vary widely, so that determining the appropriate level of monitoring becomes an important issue. In production environments, in particular, there will be some scalability limit beyond which the measurement servers start losing monitoring messages, become irresponsive, and fail to meet users' monitoring QoS requirements. In the case of OVTA, this limit is 7200 records/minute [24]. Furthermore, an increased level of monitoring also implies additional overhead at the application servers being monitored. For example, Table I shows the rise in CPU utilization due to different levels of monitoring in one of our servers running RUBiS business logic.

The basic insight from the experiments discussed above is that it is important to monitor end-to-end transaction performance, but at the same time, too much monitoring may compete with the application and other services for CPU, networking, and storage resources, thereby negatively affecting system performance. The outcome is that in order to provide QoS guarantees for monitoring, we must dynamically control monitoring itself, restricting to acceptable levels the volume of monitoring data produced and the extent to which they are analyzed.

So far, the key messages in this paper are that (i) enterprise system monitoring must itself be controlled, and (ii) that such monitoring must offer different levels of QoS. The third important insight is that to attain (i) and (ii), QoS in monitoring must be closely coupled with system utility. This is because utility directly relates monitoring and management actions to the value a business derives from its IT infrastructure. In datacenter environments, business utility ($\cup$) or revenue earned by an IT service is usually related to the monitoring cost (C) as follows:

$$\cup \quad \propto \quad a \cdot C \; - \; b \cdot C \times C$$

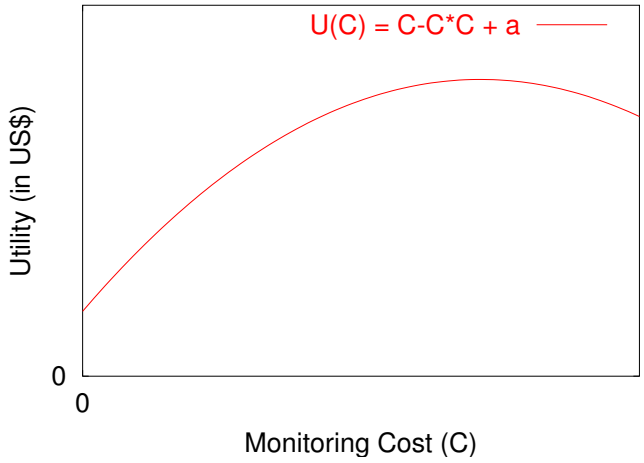

Fig. 2. Utility - Monitoring Cost Relationship

Figure 2 shows the above relationship. Monitoring cost is directly proportional to the QoS level. Utility increases up to a certain point, beyond which monitoring costs dominate. In Section IV, we experimentally verify the above relationship and show the effects of different QoS levels on the total utility achieved.

To summarize, a QoS-aware monitoring infrastructure must support the following features:

- *Flexible*: a wide variety of monitoring requirements must be accommodated, addressing the needs of different online management tasks.
- *Configurable*: runtime customizability must be supported, to match monitoring behavior and overheads to current management needs.
- *Utility-aware*: monitoring must be driven by the utility achieved by the end user application being monitored and managed.
- *Scalable*: the perturbation introduced by monitoring must be kept low.

For the QMON QoS-aware monitoring system, the next section describes how it meets the requirements articulated above.

III. QMON DESIGN AND ARCHITECTURE

QMON provides mechanisms and APIs to achieve differential quality-of-service, and it is also able to adapt to changes in requirements. Specifically, its programmable APIs expose multiple *knobs* to switch to different QoS levels and/or to tune selected parameters dynamically. These *knobs* are controlled by a policy engine driving the monitoring system's management. This paper will mostly focus on the mechanisms and APIs offered by QMON. The policy engine and suitable policies are described in [26].

The list below summarizes the basic mechanisms in QMON available to higher level policies:

- *What to monitor* (i.e., which resources, services)?

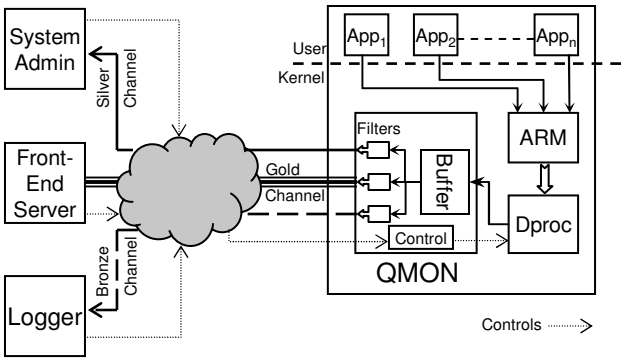

Fig. 3. QMON Architecture

- *How much to monitor* (i.e., level of detail, frequency, threshold, richness, granularity)?
- *How to deliver* (i.e., which communication transport (e.g. UDP, TCP), transport attributes (push/pull))?
- *How to filter* (i.e., pre-defined or custom filters)?
- *How to aggregate/correlate/process monitored data*?
- *How to tune/control monitoring systems*?

The resulting framework for monitoring with differential QoS has multiple advantages. First, the monitoring system can prioritize between what to monitor for which data recipient, depending upon the need and the corresponding business value or utility associated with that data and recipient. Second, monitoring costs can be controlled by quantifying it in terms of the additional utility achieved from said monitoring data. Third, given these quantifications, policies can be formulated that dynamically adapt monitoring to system changes like shifts in workload.

The basic abstraction QMON uses to achieve differential QoS is the monitoring '*channel*' (Figure 3). The channel notion is based on our earlier work on publish-subscribe middleware [14], where monitoring information is 'pushed' by data publishers (or producers) into channels, and data consumers subscribe to these channels. QMON extends this basic notion with QoS primitives that permit each channel subscriber to state its own QoS goals, to control the rate and granularity of the information flowing to it. This extension therefore, causes different channel subscribers to be treated differentially. The implementation of this functionality uses pre-defined or dynamically created channel operators – termed 'filters' – using E-Code (a language similar to C) [13] or using dynamic linking. In addition, channels can be dynamically created, deleted, and configured, the latter including the specification and use of dynamic 'attributes' associated with channels.

QMON channels enable the functionality sought from a QoS-aware monitoring system listed above. First, the association between data users and the channels that produce data is dynamic and can be changed depending on current requirements (i.e., what and how to monitor). Second, built into channels is the ability to perform the data analyses needed to provide differential monitoring, such as computing averages over multiple monitoring records (i.e., how to monitor, filter, aggregate, etc.). Third, these analyses can be dynamic (specified in the form of e-code) and can be enabled or disabled as required (i.e., how to tune/control). Finally, QoS attributes may be used to control monitoring. To address data delivery, we next present additional detail about the QMON system.

System-level resources. QMON captures system-level monitoring information via '*dproc sensors*' [1]. *Dproc* is a kernel-level monitoring toolkit for Linux-based distributed systems, such as cluster servers. The toolkit provides a single uniform user interface available through /*proc*, which is a standard feature of the Linux operating system. *Dproc* extends the local /*proc* entries of each of the cluster machines with relevant information from all other participating nodes within the cluster. Kernel-level data capture permits *dproc* to capture the joint behavior of multiple system resources. Kernel-level communications enable the exchange of monitoring information across participating nodes with predictable delays. Toward this end, *Dproc* uses a binary messaging system with out-of-band meta-information and very low marshalling overhead, which makes it suitable for use in high end enterprise systems. Despite its kernel-level operation, *dproc* can be dynamically extended with new monitoring functionality. Plug-and-play monitoring modules can be added at run-time to permit *dproc* to deal with new devices or resources and/or to offer new performance models of resources to applications.

ARM Extension to *dproc*: End-to-End performance measurement is necessary for effective management of enterprise applications. QMON uses the ARM (Application Response Measurement) standard [3] to define a common way to describe transaction-level information that can be analyzed to detect SLA violations, bottlenecks and other performance problems. ARM agent running on each application node (e.g., web server, application server, and database server) collects and summarizes the end-to-end performance for transactions starting from this machine. An ARM agent can provide data about each individual transaction instance (i.e., trace) or summarize data across many transaction instances (e.g., sample). There may be additional ARM servers that collect and correlate monitoring data from ARM agents on different application components. Many commercial applications have been instrumented with ARM, particularly in J2EE, such as the IBM WebSphere Application Server, and IBM DB2. Plug-ins have also been written for widely used components, such as the Apache HTTP server and Microsoft's Internet Information Services. As a result, the application's EJB programs and Servlets are measured without the application source code being changed.

For precise resource information, we extend the *dproc* monitoring system to gather and analyze ARM metrics. Application-level ARM instrumentation invokes the *dproc* API to log information about their transactions. *Dproc* logs and marks them with their associated resource usage, such as the CPU time consumed during the duration of the transaction. This information is very useful for accounting purposes and

```
typedef struct _ARM{
    int type, ID, start_time, end_time;
} ARM;

int ARM_filter{
    int i,sum[TOTAL_TYPES],count[TOTAL_TYPES];

    for (i = 0; i < TOTAL_TYPES; i++){
        sum[i] = 0;
        count[i] = 0;
    }
    for (i = 0; i < input.arm.count; i++){
        type = input.arm.data[i].type;
        sum[type] = input.arm.data[i].end_time -
                    input.arm.data[i].start_time;
        count[type] ++;
    }
    for (i = 0; i < TOTAL_TYPES; i++)
        output.avg[i] = sum[i] / count[i];

    return SEND_FILTERED_DATA;
}
```

Fig. 4. ARM Filter E-Code

to detect malicious or faulty behavior. *Dproc* maintains a pool of internal buffers in which ARM statistics are stored. Application or services can register to receive ARM data and subscribe to a standard filter or specify custom ones. Figure 4 shows an example configuration of ARM data and a *dproc* filter that calculates the average time taken by different types of transactions.

IV. Experimental Evaluation

We evaluate QMON with a set of microbenchmarks and with application-level measurements. While QMON can support arbitrary monitoring channels with rich associated methods of QoS, in the remainder of this paper, we experiment with a simple notion of QoS widely used in existing exterprise platforms (e.g., in IBM's Websphere XD):

Gold Channel: monitors at higher priority than any other types of channels. It carries dproc exported system usage information (i.e., CPU, memory, network, block I/O) and application-level performance data in the form of ARM transactions. Monitoring data is buffered and broadcasted to subscribers of gold channels every two seconds. This kind of QoS is required by the front-end server scheduler in a multi-tier web service for making online scheduling and dispatching decisions.

Silver Channel: monitor at a lower granularity than gold channel, transporting system resource information at an interval of 30 seconds. Instead of sending raw ARM data, it condenses them by calculating their mean and variance and publishes condensed information every 1 minute. Figure 4 shows a simplified version of the filter applied to ARM data in silver channel.

Bronze Channel: provides best-effort service, collecting the same data as the silver channel, however, publishes it every 5 minutes. This kind of QoS is generally required for offline auditing and analyses purposes.

The remainder of this section first uses microbenchmarks to assess the ability of QMON to provide the different levels of QoS required for gold, silver, and bronze service levels. Then, these notions are used to monitor and manage a representative web services application.

A. Microbenchmarks

The first set of experiments evaluate the overheads associated with the three different kind of channels described above. All the experiments are performed on a cluster of 16 Intel Xeon 2.8 GHz nodes, each with 512KB cache and 512MB RAM, and connected via a 1 Gbit LAN. Each node runs the RedHat Linux 9 (kernel version 2.4.19). The monitoring infrastructure of the QMON system is implemented as loadable kernel modules.

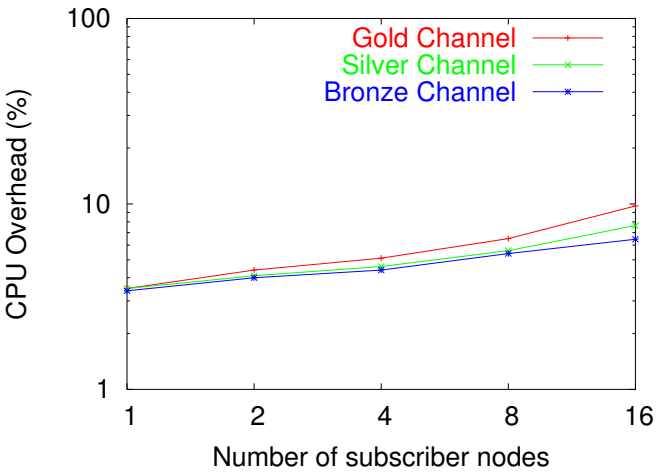

Fig. 5. Microbenchmark: Publisher CPU Overhead

We run QMON on a node (publisher) and vary the number of subscribers connecting to that node. The aim of this experiment is to calculate the scalability of the QMON system by measuring the CPU overhead at the node that publishes monitoring data. We create a *gold channel* with the specification as described earlier and let all the nodes subscribe to it and receive monitoring information. A simple script generating 100 ARM messages every second runs at user-level to mimic the behavior of an actual ARM agent (like OVTA). We run this setup for over a minute and capture CPU usage with the *sar* utility. The same experiment is repeated for *silver* and *bronze* channels. Figure 5 shows the results of this experiment. The overhead is less than 10% even when the number of subscribers is 16. There is a very slight difference between the overheads of the three channels. This is because the instrumentation overhead remains the same in the three cases. The difference lies in the rate and the granularity of the data that is broadcast to the subscribers. The high bandwidth

low latency link used by *dproc* together with its in-kernel data analysis contribute to the system's low overheads.

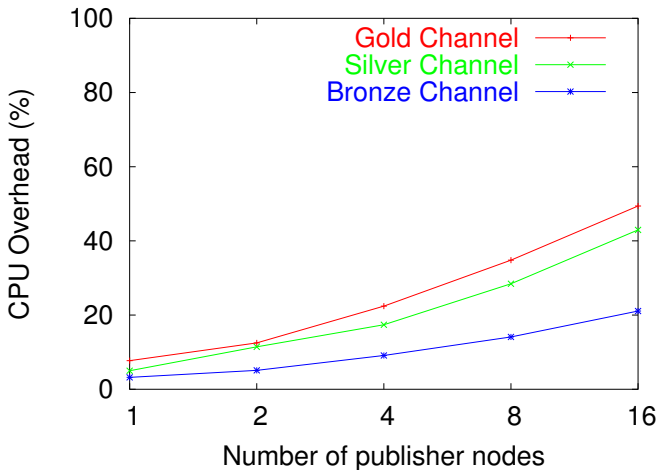

Fig. 6. Microbenchmark: Subscriber CPU Overhead

The next experiment evaluates the cost of receiving monitoring data from publishers. The setup remains the same as in the earlier experiment. The subscriber (or the *analyzing node*) registers with up to 16 different publishers and starts receiving information from them periodically. A user-level script reads all data from system-level QMON every 30 seconds, emulating the behavior of a GUI client used by the system administrators to analyze enterprise performance. As the number of publishers increases, the amount of data to be processed also increases. The experiment establishes the fact that such a user-level client can consume substantial CPU cycles copying data from the monitoring channels via QMON to user-space. Further, there is a significant difference in CPU usage between different channels because of the different quality(/quantity) of data carried by them.

The experiments in this section demonstrate QMON's ability to provide different levels of QoS in monitoring and its ability to switch between them in a cost-efficient manner. Although the cost at the measurement node remains relatively constant with the change in number of subscribers (see Figure 5), the cost at the analyzing node increases rapidly with the increase in the number of publishers (see Figure 6). This is where QMON's flexibilty becomes important, as the analyzer can choose the type of monitoring information it wants to receive and the QoS associated with them so that it can scale to larger number of messages and more complex enterprise scenarios.

B. Application Benchmark

As discussed earlier, we employ RUBiS to illustrate the efficacy of QMON. We use the Servlets version of RUBiS with Apache 2.0.40 web server at the front end, two Jakarta Tomcat 5.5.9 servelet server, and a database server running MySQL 4.1.14. All servers are hosted on dual Intel Xeon 2.8 GHz servers with 512KB cache and 4GB RAM, and connected via 1 Gbit ethernet. Each of the machines runs RedHat Linux 9.0 (kernel version 2.4.19) with QMON extensions.

The servers run in their default configuration, except for the following settings:

- *MaxSpareServers* of the Apache web server is increased to 50 so that the server doesn't spend too much time in forking threads at the beginning of each experiments, thereby affecting our readings;
- Initial Heap Size for the Servlet Container ($-Xms$): 128MB;
- Maximum Heap Size for the Servlet Container ($-Xmx$): 768MB; and
- Stack Size of each Servlet's Thread ($-Xss$): 128KB.

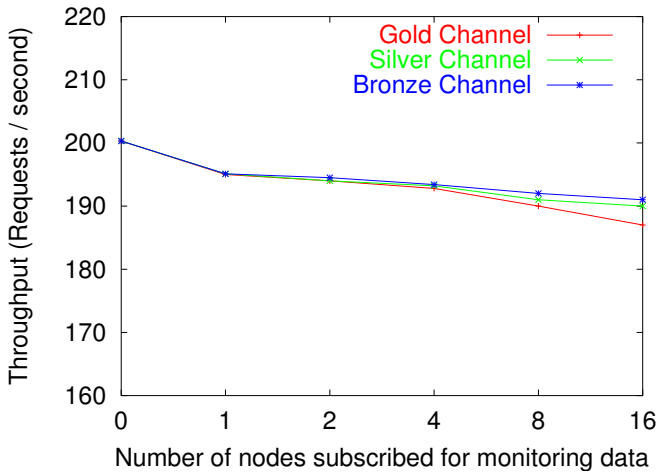

Fig. 7. Throughput degradation due to QMON

The first experiment evaluates the change in performance of RUBiS due to QMON. We test the raw throughput obtained from RUBiS by running a stream of *user registration* requests, first without QMON, and then enabling QMON and publishing the monitoring data to other nodes. Figure 7 shows the result of this analysis. As soon as we enable QMON, there is a 3% reduction in performance. The degradation is just 6% even when the number of nodes subscribed to receive monitoring information increases to 16. This is because of the low-level kernel implementation of QMON, which maintains a list of all subscribers inside the kernel and does a fast network transfer to all of them without copying any data from the user level for each transfer.

The front end server has to perform request scheduling and dispatching, the purpose of which is to ensure load balancing and provide quality of service. For our evaluation, we use a simple black-box scheduling algorithm called *DWCS* [30]. DWCS was originally developed for streaming multimedia applications, to schedule their processes and/or perform message scheduling [29]. In this paper, we use it in enterprise domain where different workloads must be multiplexed in a shared utility infrastructure (like a multi-tier web service). These workloads are often associated with some performance goals (like minimum throughput or best response time) and

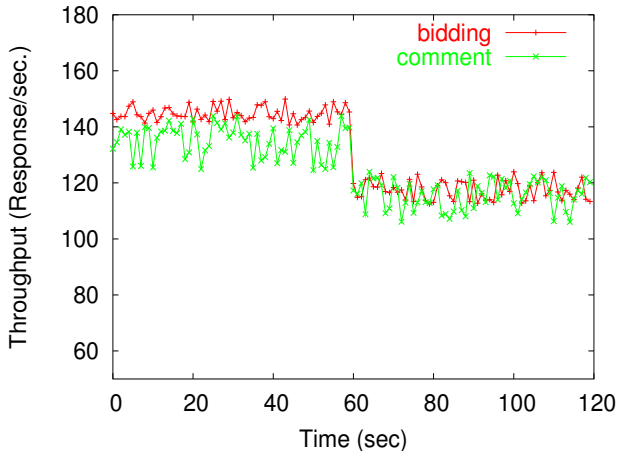

Fig. 8. Throughput with QMON disabled

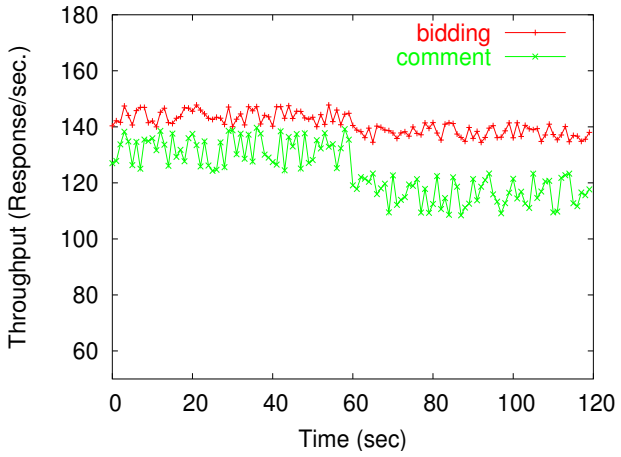

Fig. 9. Throughput with QMON enabled

may have certain real-time requirements, the latter typically expressed in the form of SLAs. For example, a *bidding* request in an online auction site like RUBiS has real-time deadlines, while a *comment* posted by a user has a less stringent deadline.

We apply DWCS to schedule two different request classes in the RUBiS with QMON disabled. These requests are generated using *httperf* [20] on a separate client machine with the same configuration as other server machines. The *bidding* request is computation intensive and consumes substantial CPU at the servlet server processing it. In contrast, the *comment* request generates significant network traffic. The scheduler runs on the same node as the client, and request dispatching is facilitated by prefixing the request's URL path with the appropriate servlet server's name. The Apache server is configured to multiplex the requests to the different backend server depending on these prefixes.[1]

The purpose of the experiments described below is to demonstrate the ability of QMON to deliver QoS in monitoring, and next, to show the importance of monitoring QoS when delivering improved utility to end user applications.

1) Application Interference: Consider a scenario in which RUBiS components are operating correctly, but the utility derived from their operation is suddenly diminished. One cause for such a reduction in utility is undue resource consumption by a foreign application running on a machine used by a RUBiS component. To maintain high levels of utility, the enterprise must quickly re-schedule the foreign application to a different machine. This requires monitoring support that can promptly identify the foreign application, the fact that it consumes the resources required by RUBiS, and then notify administrators or higher level policy engines to take appropriate actions. Undue delays in such actions result in queue buildups and similar issues with backlogged requests that can quickly spread to other components of a distributed enterprise application [18].

[1]The scheduler could have been implemented in the front-end web server, but for simplicity, we choose to emulate all client sessions with *httperf* and schedule their requests with DWCS on the same machine.

We demonstrate the effect of application interference by creating 60 client sessions, half of which are high priority *bidding* requests, the other half being low priority *comment* requests. Each request class has a *Poisson* arrival distribution with a mean rate equal to 150 requests/sec. In the middle of the experiment, a perturbation is introduced by starting four linpack processes.

The performance of both classes is reduced by more than 15% (Figure 8). In Figure 9, we enable QMON, which resulted in the higher priority *bidding* requests exhibit only a small drop in performance because these requests are routed to the server that is lightly loaded. Online monitoring with QMON, therefore, provides the resource-awareness required for scheduling to attain high utility, that is, better service for bidding vs. other requests. In comparison, a non-aware algorithm simply performing round-robin scheduling does not perform well.

2) Component Misbehavior: QoS in monitoring refers much more than just monitoring latency or delay. Consider applications that exhibit temporary misbehavior, perhaps due to garbage collection or poison messages [18]. This results in a sudden reduction in performance of a particular component, from which it recovers after some time, or its behavior is permanently affected. An example is the behavior observed in our partner's Airline Reservation system (see Section II), where certain components exhibit reduced performance from which they recover after some time. A concrete illustration of this behavior is realized with a modified RUBiS application, which delays request processing by a few milliseconds every alternate minute such that the first tomcat servlet server delays every odd minute and the second server delays every even minute. Both recover in the subsequent minute.

Monitoring QoS in scenarios like these refers to the granularity of monitoring information. Low QoS means that a system is observed over a long time period, and administrators or policies receive only occasional reports on average system behavior. In that case, erratic behaviors like those described

above would not be detected. Stated more precisely and drawing a parallel from the classical *Nyquist-Shannon sampling theorem*, it is apparent that in the RUBiS case, monitoring must be done at least once every minute to detect the emulated server misbehavior.

We study three scenarios. In the first, the front-end web server subscribes to a *bronze channel* to receive monitoring information from the two tomcat servers. In the second, it subscribes to a *silver channel*. In the third scenario, we define a new type of *platinum channel* which offers a level of QoS in which details are collected at a granularity finer than the *gold channel* described earlier. That is, it not only collects application-level end-to-end ARM transaction information, but it also captures packet level details and sends digests to the scheduler every 100 milliseconds. The workload used in this experiment consists of a stream of RUBiS requests from two different clients and has a *Poisson* arrival distribution with a mean rate equal to 150 requests/sec. Concerning utility, Client 1 pays twice the price than Client 2, according to the following utility formulation:

$$\cup \propto \frac{1}{latency}, \text{ and } \cup_{client1} = 2 \times \cup_{client2}$$

Figure 10 shows the total utility obtained by processing requests from two clients for 10 minutes. The utility is higher when monitoring is done more frequently via the *silver channel*. The *bronze channel* does monitoring at lower rate (every 5 minutes) because of which it fails to capture the misbehavior of the two servers (because the average latency over the two minute window remains almost same). The *silver channel*, on the other hand, publishes performance information every 30 seconds, which permits the front-end web-server to detect the change in latency of the two backend servers. This enables the front-end to make more appropriate request routing decisions. Basically, it dispatches the requests from Client 1 to the server with lower delay because it earns more revenue (i.e., higher utility). Further, although the utility of Client 2 goes down by 11%, the utility of Client 1 goes up by 30%, improving total utility by 16.7%.

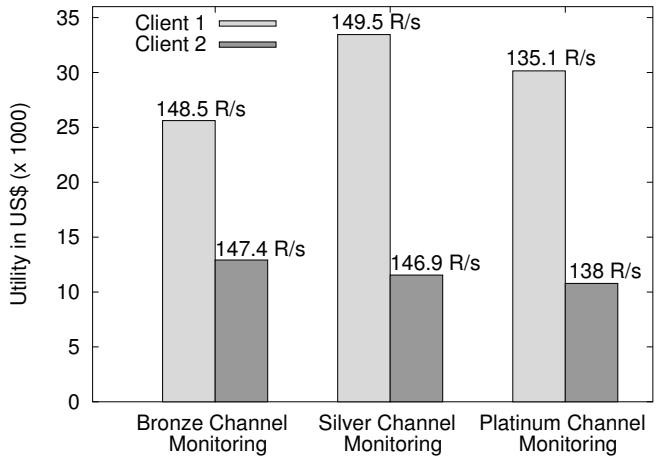

Fig. 10. Change in Utility due to QoS in monitoring

Note that the average throughput (i.e., responses per second) remains the same for both clients when monitoring is done with either silver or bronze channels. This is because the capacity of the system is larger than the workload. As latency is the major factor in determining utility, the details obtained from the *silver channel* helped the scheduler route judiciously. However, when the scheduler switches to a *platinum channel*, throughput goes down because of increased monitoring overheads. This decrease in throughput is also reflected in the response time of the servers. Hence, total utility is reduced as compared to the *silver channel*. These results demonstrate two important facts about online monitoring: (1) fine-grain frequent monitoring is necessary to achieve higher utility and to adapt to time-varying resource availability, but (2) overly aggressive monitoring can have negative effects, including reducing the overall utility derived from the enterprise application.

V. Lessons Learned

QoS imposes unique requirements on monitoring: in terms of classes of users, time granularity, and the amounts of monitored data. Since there will always be tradeoffs between the quality of data collected and the costs involved, our work attempts to quantify the cost of providing QoS in monitoring from the business perspective.

The first lesson from QMON is that **even a carefully designed monitoring system will still impose costs** that can notably affect the maximum raw throughput of applications or services. For instance, QMON microbenchmarks establish that despite low data collection and transmission overheads in our uses of QMON, there are still substantial overheads experienced by monitoring data recipients, causing scalability issues. This is one motivation for offering enterprise users different levels of QoS in monitoring. Another motivation is that there is a need for different levels of QoS in terms of the quality of monitoring data captured by the system and provided to applications. There are no issues with scalability in the experiments shown in Sections IV-B.1 and IV-B.2, which use only two application servers, but the front-end scheduler will at different times require different levels of data granularity, in order to scale to large number of application servers.

Monitoring must be programmatically configurable in order to support QoS. As discussed earlier, traditional monitoring systems rely on the system administrators to configure different parameters. With the increasing trend towards automated management and with the increase in the complexity of monitoring itself, the process of manual configuration becomes acutely slow, costly, and error-prone. Furthermore, dynamic changes in users' requirements, system workloads, and platform resources require that the monitoring system adapt itself to those changes automatically. Toward these ends, QMON provides rich methods for creating, deleting, and configuring "*Monitoring Channels*". The use of programmable APIs make it easy for applications to switch between multiple QoS levels

of monitoring in order to receive the required '*quality*' of data and maintain low overheads.

QoS in monitoring should be closely coupled with business objectives (i.e., application utility). This is most evident from the results in Section IV-B.2, where infrequent updates of monitoring information and aggregation over large time windows fail to provide sufficient levels of detail to permit appropriate request scheduling actions. By tuning the granularity of monitoring, undesirable changes in component performance could be recognized. While this increases the raw cost of monitoring at back-end application servers, the resulting improvements in scheduling at the front-end still increase total utility. The lesson is that rather than attempting to minimize monitoring overhead for individual subsystems or components, monitoring should be managed so as to maximize overall utility (or business revenue).

Finally, although this paper shows positive results about the importance of QoS in monitoring for automated management, it remains up to future work to better quantify the exact relationship between the cost of QoS in monitoring and the utility achieved from an enterprise application. That is, this paper's simple bronze, silver, gold characterization of monitoring QoS should be refined to take into account the large variety of metrics capturing QoS in monitoring, ranging from data volume, to data precision, to methods of data delivery, etc.

VI. Related Work

We are not aware of other research on system monitoring that specifically focuses QoS issues. As a result, past work has not delivered the programmable APIs and frameworks for monitoring that support dynamic monitoring reconfiguration or provide QoS guarantees for enterprise-scale applications and systems. Further, the formulation of QoS differs for monitoring compared to past work in the multimedia [4] and more recently, in the mobile domain [9], where QoS is expressed in terms of metrics like delay, jitter, bandwidth, throughput, etc. In the enterprise domain, monitoring poses additional challenges, including taking into account its costs (i.e., overheads), the tradeoffs between the quality of monitoring data generated and the perturbation introduced in the system being monitored, and the fact that QoS is determined not only by data delivery, but also by data generation and the analyses applied.

There is rich prior work in the area of distributed monitoring. Ganglia [19] is a scalable distributed monitoring system for high performance computing systems. MonALISA [21] provides a distributed monitoring service based on a scalable dynamic distributed architecture. ACME [23] is a flexible infrastructure for Internet-scale monitoring, analysis, and control. Compared to these systems, our work differs in two key respects. First, since these systems are designed mainly for monitoring distributed systems and Grids, they do not address the requirements of enterprise monitoring, such as dealing with service level SLOs, providing end-to-end transaction information, performing real-time and dynamic service monitoring, and supporting the interactions between the monitoring system and online management components (e.g., a request scheduler). In particular, none of them address real-time monitoring with QoS guarantees. Second, these systems focus mainly on data collection, delivery, and scalability, but they do not address the dynamism in monitoring systems in terms of changes of users' monitoring requirements and changes in monitored environments. As a result, they do not provide programmable APIs and dynamic mechanisms to support runtime monitoring configuration.

HP and IBM have developed their own monitoring solutions in the enterprise domain. HP's OpenView monitoring products [22] (performance agent, transaction analyzer, network node manager, performance manager, etc.) implement a flexible distributed monitoring solution for enterprise management. IBM's Tivoli monitoring [27] is an enterprise-class monitoring solution, which monitors the availability of the IT infrastructure, end-to-end, across distributed and host environments. Both provide rich configurability to control monitoring, such as what to monitor, how much to monitor, etc. However, such configuration must be done manually. This makes it difficult to support dynamic reconfiguration, adapt to changes in the computing environment, or perform the custom monitoring needed to deal with complex behaviors in enterprise applications and environments.

VII. Conclusions and Future Work

Runtime monitoring is key to the effective management of enterprise and high performance applications. At the application and middleware level, service execution must be continuously monitored to ensure that the service level objectives defined by administrators are continuously met. At the system level, service resource usage must be monitored, to ensure sufficient resources for meeting SLOs (i.e., resource provisioning and capacity planning), to detect and deal with system bottlenecks due to dynamic service and platform behaviors, and to enable dynamic optimization or weaker properties like performance isolation.

This paper demonstrates the need for explicit support of quality of service (QoS) in monitoring for enterprise systems. QoS must be dynamically configurable to obtain a balance between monitoring overheads and the improvements in application utility derived from online monitoring and management. Further, we describe the design and architecture of a QoS-aware monitoring system called QMON. QMON provides the abstraction of "*Monitoring Channels*" as a means to implement differential QoS in monitoring. QMON is evaluated on a deployment of a multi-tier web service benchmark, and evaluations shown that multiple and different levels of QoS in monitoring are necessary to obtain high application utility. Results also demonstrate that insufficient or excessive granularity of monitoring are both detrimental to overall system utility. The key is finding a balance. A configurable, flexible monitoring system providing guaranteed QoS is a first step toward that goal.

Our future work will make more sophisticated use of QoS in QMON, generalizing our current relatively simple bronze,

silver, golden notions of QoS. In addition, a clearer linkage will be established between monitoring QoS and overheads and the utility derived from monitoring, using formal techniques to quantify and link both. Another interesting direction of our research is one that extends the predictable methods for system-level monitoring designed in our work on *dproc* to also capture end-to-end application behaviors.

REFERENCES

[1] S. Agarwala, C. Poellabauer, J. Kong, K. Schwan, and M. Wolf, "Resource-Aware Stream Management with the Customizable dproc Distributed Monitoring Mechanisms," in *Proceedings of the 12th IEEE International Symposium on High Performance Distributed Computing (HPDC-12), Seattle, Washington*, June 2003, pp. 250 – 259.

[2] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: An Extensible System for Design and Execution of Scientific Workflows," in *16th International Conference on Scientific and Statistical Database Management (SSDBM'04)*, 2004, p. 423.

[3] "Systems Management: Application Response Measurement (ARM)," Open-Group Technical Standard, Catalog number C807, ISBN 1-85912-211-6, July 1998, http://www.opengroup.org/products/publications/catalog/c807.htm.

[4] C. Aurrecoechea, A. T. Campbell, and L. Hauw, "A survey of QoS architectures," *Multimedia Systems*, vol. 6, no. 3, pp. 138 – 151, May 1998.

[5] N. Bhatti, A. Bouch, and A. Kuchinsky, "Integrating User-Perceived Quality into Web Server Design," in *Proceedings of the 9th International World Wide Web Conference, Amsterdam, Netherlands*, May 2000, pp. 1–16.

[6] N. Bhatti and R. Friedrich, "Web server support for tiered services," *IEEE Network*, vol. 13, no. 5, pp. 64–71, September 1999.

[7] J. T. Buck, S. Ha, E. A. Lee, and D. G. Messerschmitt, "Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems," *Int. Journal of Computer Simulation, special issue on "Simulation Software Development*, vol. 4, pp. 155–182, April 1994.

[8] E. Cecchet, J. Marguerite, and W. Zwaenepoel, "Performance and Scalability of EJB Applications," in *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Seattle, Washington, USA*, November 2002, pp. 246–261.

[9] S. Chakrabarti and A. Mishra, "QoS issues in ad hoc wireless networks," *Communications Magazine, IEEE*, vol. 39, no. 2, pp. 142 – 148, February 2001.

[10] S. E. Chodrow, F. Jahanian, and M. Donner, "Run-time monitoring of real-time systems," *Monitoring and debugging of distributed real-time systems*, pp. 103–112, 1995.

[11] I. Cohen, J. S. Chase, M. Goldszmidt, T. Kelly, and J. Symons, "Correlating instrumentation data to system states: A building block for automated diagnosis and control," in *6th USENIX Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA*, December 2004, pp. 231–244.

[12] Y. Diao, F. Eskesen, S. Froehlich, J. L. Hellerstein, L. Spainhower, and M. Surendra, "Gnereic Online Optimization of Multiple Configuration Parameters with Application to a Database Server," in *Self-Managing Distributed Systems, 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), Heidelberg, Germany*, October 2003, pp. 3–15.

[13] G. Eisenhauer, "Dynamic Code Generation with the E-Code Language," Georgia Institute of Technology, College of Computing, Tech. Rep. GIT-CC-02-42, July 2002.

[14] G. Eisenhauer *et al.*, "Publish-subscribe for High-performance Computing," *IEEE Computing*, vol. 10, no. 1, pp. 40–47, January/February 2006.

[15] J. L. Hellerstein, Y. Diao, S. Parekh, and D. Tilbury, Eds., *Feedback Control of Computing Systems*. Wiley-Interscience, 2004.

[16] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41 – 50, January 2003.

[17] C. R. Lumb, A. Merchant, and G. A. Alvarez, "Facade: Virtual Storage Devices with Performance Guarantees," in *Proceedings of the USENIX FAST '03 Conference on File and Storage Technologies, San Francisco, California, USA*, March 2003.

[18] M. Mansour and K. Schwan, "I_RMI: Performance Isolation in Service Oriented Architectures," in *Proceedings of the 6th ACM/IFIP/USENIX Int'l Middleware Conference (Middleware 2005)*, November 2005.

[19] M. L. Massie, B. N. Chun, and D. E. Culler, "The ganglia distributed monitoring system: Design, implementation, and experience," *Parallel Computing*, vol. 30, no. 7, pp. 817–840, July 2004.

[20] D. Mosberger and T. Jin, "httperf: A Tool for Measuring Web Server Performance," *Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, December 1998, originally appeared in Proceedings of the 1998 Internet Server Performance Workshop, June 1998, 59-67.

[21] H. B. Newman, I. C. Legrand, P.Galvez, R. Voicu, and C. Cirstoiu, "MonALISA: A Distributed Monitoring Service Architecture," in *Conference for Computing in High Energy and Nuclear Physics(CHEP), La Jolla, California*, March 2003.

[22] "HP OpenView," http://www.openview.hp.com.

[23] D. Oppenheimer, V. Vatkovskiy, H. Weatherspoon, J. Lee, D. A. Patterson, and J. Kubiatowicz, "Monitoring, Analyzing, and Controlling Internet-scale Systems with ACME," EECS Department, University of California, Berkeley, Tech. Rep. UCB/CSD-03-1276, 2004.

[24] "HP OpenView Transaction Analyzer performance and scalability Guide," available from http://www.managementsoftware.hp.com/products/tran/.

[25] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for QoS management," in *IEEE Real-Time Systems Symposium*, December 1997, pp. 298 – 307.

[26] C. Shankar, V. Talwar, S. Iyer, Y. Chen, D. Milojicic, and R. Campbell, "Specification-enhanced policies for automated change management of it systems," in *In submission*, 2006.

[27] "IBM Tivoli Monitoring." http://www-306.ibm.com/software/tivoli/products/monitor/.

[28] J. S. Vetter and D. A. Reed, "Real-Time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids," *International Journal of High Performance Computing Applications*, vol. 14, no. 4, pp. 357–366, 2000.

[29] R. West, I. Ganev, and K. Schwan, "Window-Constrained Process Scheduling for Linux Systems," in *Proceedings of the 3rd Real-Time Linux Workshop, Milan, Italy*, November 2001.

[30] R. West, Y. Zhang, K. Schwan, and C. Poellabauer, "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers," *IEEE Transactions on Computers*, vol. 53, no. 6, pp. 744–759, June 2004.

[31] J. Wilkes, J. Mogul, and J. Suermondt, "Utilification," in *Proceedings of the 11th ACM-SIGOPS European Workshop, Leuven, Belgium*, September 2004.

[32] R. Wolski, N. Spring, and J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems*, vol. 15, no. 5-6, pp. 757–768, October 1999.