SYSTEM SUPPORT FOR END-TO-END PERFORMANCE MANAGEMENT

A Thesis
Presented to
The Academic Faculty

by

Sandip Agarwala

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
August 2007

SYSTEM SUPPORT FOR END-TO-END PERFORMANCE MANAGEMENT

Approved by:

Prof. Karsten Schwan, Committee Chair
College of Computing
Georgia Institute of Technology

Prof. Karsten Schwan, Advisor
College of Computing
Georgia Institute of Technology

Prof. Mustaque Ahamad
College of Computing
Georgia Institute of Technology

Dr. Dejan Milojicic
Hewlett Packard Labs

Prof. Santosh Pande
College of Computing
Georgia Institute of Technology

Prof. Calton Pu
College of Computing
Georgia Institute of Technology

Date Approved: 9 July 2007

To my mother and father

ACKNOWLEDGEMENTS

This thesis would not have been possible without the support and advice from many individuals. I would like to express my gratitude to all of them.

First and foremost, I would like to thank my advisor Karsten Schwan for his support and guidance all throughput my Ph.D. years. I am grateful to him for providing lot of flexibility and resources to pursue my research interests. His tireless efforts to help his students in every possible way, inspite of his busy schedule, always amazed me. I owe every bit of my doctoral career to him.

I would like to acknowledge the other members of my thesis committee. I am grateful to Dejan Milojicic for his timely advice and support. His feedbacks helped me to think critically and improve the presentation of this thesis. Prof. Mustaque Ahamad, Prof. Santosh Pande, and Prof. Calton Pu, provided insightful comments that helped me improve my work.

I would also like to thank my friends at Georgia Tech for sharing so many joyous and unforgetful moments with me. They were one of the major sources of inspiration and helped me survive my Ph.D. years. Last but not the least, I would like to extend my deepest thanks to my family members, especially my parents, for their love, sacrifice and encouragement to make it through. This work is dedicated to all of them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Distributed systems are becoming increasingly complex, in part because of the prevalent use of web services, multi-tier architectures, and grid computing, where dynamic sets of components interact with each other across distributed and heterogeneous computing infrastructures. An important problem in this domain is to understand the runtime behaviors of these networked applications and systems. For an enterprise, understanding the end-to-end dynamic behavior of its IT infrastructure, from the time requests are made to when responses are generated and finally, received, is a key prerequisite for improving application response, to reduce failure recovery times, or to avoid undesired effects like congestion. It is also a prerequisite for compliance checking with legal requirements or to meet service level agreements (SLAs), which can have direct legal or financial implications.

End-to-end behavior understanding faces multiple barriers. First, the large diversity of applications routinely used in the enterprise, the presence of legacy components, and source code inaccessibility make it difficult or perhaps, impossible, to instrument distributed applications, despite new industry instrumentation standards like ARM. Second, effective management requires end-to-end measurement that scales, both in the number of metrics collected about specific components and the number of components or machines involved in the application. Simple metrics like average CPU load, network bandwidth, number of tasks completed, etc. are often insufficient for root-cause analysis. Third, application evolution can disable or invalidate existing instrumentation, leading to erroneous measurement results, and this also holds for rapidly changing enterprise middleware, which are driven more by new features and functionality than by precise instrumentation.

This dissertation introduces, implements, and evaluates the novel concept of “Service Paths”, which are system-level abstractions that capture and describe the dynamic dependencies between the different components of a distributed enterprise application. Service

paths are dynamic because they capture the natural interactions between application services dynamically composed to offer some desired end user functionality. Service paths are distributed because such sets of services run on networked machines in distributed enterprise data centers. Service paths cross multiple levels of abstraction because they link end user application components like web browsers with system services like http providing communications with embedded services like hardware-supported data encryption. Service paths are system-level abstractions that are created without end user, application, or middleware input, but despite these facts, they are able to capture application-relevant performance metrics, including end-to-end latencies for client requests and the contributions to these latencies from application-level processes and from software/hardware resources like protocol stacks or network devices. Unique service paths are created for different classes of application-level requests, thus making the notion of request class the basic quantum of enterprise management they support. Finally, service paths are constructed by software modules embedded into operating systems on participating machines. These modules dynamically measure and analyze program behavior.

Beyond conceiving of service paths and demonstrating their utility, this thesis makes three concrete technical contributions. First, we propose a set of signal analysis techniques called “E2Eprof” that identify the service paths taken by different request classes across a distributed IT infrastructure and the time spent in each such path. It uses a novel algorithm called “pathmap” that computes the correlation between the message arrival and departure timestamps at each participating node and detect dependencies among them. A second contribution is a system-level monitoring toolkit called “SysProf”, which captures monitoring information at different levels of granularity, ranging from tracking the system-level activities triggered by a single system call, to capturing the client-server interactions associated with a service paths, to characterizing the server resources consumed by sets of clients or client behaviors. SysProf operates enterprise-wide in that it can track the application- and system-level activities and associated resource usage across the multiple machines used in typical enterprise infrastructures. The third contribution of the thesis is a publish-subscribe based monitoring data delivery framework called “QMON”. QMON offers high levels of predictability for service delivery and supports utility-aware monitoring while also able to differentiate between different levels of service for monitoring, corresponding to the different classes of SLAs maintained for applications.

Service Paths and the E2EProf methods implementing them are shown useful for and are evaluated with multiple enterprise-class applications, which include a high performance shared network file service, a multi-tier auctioning web site and an Operational Information System (OIS) used by our industrial partners. For these applications, service paths are used to automatically detect request paths and then find performance bottlenecks associated with those paths. In addition, they are used to enforce application-level SLAs by adapting the middleware's or application's scheduling and dispatching behavior based on the resources consumed by different request classes. Service paths can also assist in dynamic application tuning and provisioning, by using them when making system management decisions like adding servers, adjusting the number of threads used in applications, changing buffer sizes, etc. Finally, we demonstrate the efficacy of the general approach to runtime analysis taken by our work by first using E2EProf to isolate problematic nodes in a service path and then use the additional and relatively more intrusive capabilities of SysProf to "zoom" into those nodes to perform micro-level analysis and identify the exact workloads, application components, and/or resources causing the performance problem.

CHAPTER I

INTRODUCTION

1.1 Background

Web services, multi-tier software architectures, virtualization, and grid computing [10, 35, 40] are enabling distributed applications in which dynamic sets of components freely interact across distributed and heterogeneous computing infrastructures. Given such applications' complex, dynamic nature, it is almost impossible to determine whether they are robustly delivering services to end users. Yet human operators must be able to track dynamic program behavior. In a datacenter environment, for instance, understanding the end-to-end behavior of certain IT subsystems, from the time requests are made to when responses are generated and finally, received, is a key prerequisite for improving application response, for guaranteeing certain levels of performance, or for meeting service level agreements (SLAs).

Similarly, in distributed scientific endeavors, end users require remote experiments to be monitored and/or controlled with small delays, but difficulties arise from the potential data floods implied by such monitoring and the inability of operating or file systems to guarantee stable behaviors for large-scale I/O [69]. Finally, in all such settings, system virtualization is creating new opportunities for dynamism in system behavior, by enabling additional methods for sharing and consolidating underlying platform resources. In summary, (1) understanding the extent to which applications are meeting their *defined quality needs* is critical when end users rely on well-functioning IT infrastructures, and (2) building on such understanding, equally important are automated methods for pinpointing potentially problematic subsystems and then, for coping with the effects of such problems.

While feature-rich distributed programming infrastructures have made it possible to quickly construct complex distributed applications for unique purposes or specific end user needs, additional progress is needed in the development of technologies that help us recognize or understand potentially problematic program behaviors [32], control them, and/or prevent their occurrence. A simple example is the incidence of what one of our industry collaborators termed 'poison messages' [61] observed in one of their IT subsystems. Their occurrence would cause undue loads on a certain subsystem, resulting in queue buildups, which would in turn cause other subsystems to slow down. Undesirable outcomes include missed events, floods of failure notifications concerning missed deadlines, etc. A more complex behavior observed in another setting and probably due to unforeseen application/OS interactions is one in which certain sequences of requests (i.e., sets of 'tainted messages' [62]) cause undue server slowdown, eventually requiring server restarts and thereby leading to losses in server farm processing capacity. In the scientific domain, similar issues arise, such as when a 'tightly programmed' simulation code running on a supercomputer interacts with a commercial operating system [77] or a general purpose data analysis package, a specific issue for the latter being its difficulty in coping with the immense volumes of data a supercomputer is able to produce [70]. In summary, it is evident that the inability to understand the runtime behavior of distributed applications can degrade or threaten their *robustness*. Further, when applications have *defined quality needs*, which means that they must consistently deliver their services within specific constraints dictated by their use, then this lack of behavior robustness directly impacts their core ability to function.

A key problem in this domain is to understand the runtime performance behaviors of these highly distributed, networked applications and systems, to better manage system assets or application response and/or to reduce undesired effects like congestion. For instance, even the processing of a simple web request may generate intricate interactions between the different web components that may span a multitude of machines. Problems in such systems, therefore, are difficult even for experts to diagnose, particularly when it comes to understanding their root causes. Different classes (SLA class, for example) of clients may have different requirements, and the business impact due to inordinate delay or loss of a particular client's request may be substantial. The average cost of downtime either due to outright outage or due to service degradation is about US$125,000 per hour [1]. It is critical, therefore, to pinpoint the sources of performance problems experienced by applications.

---

[1] IDC #31513, July 2004

Multiple technical issues make it difficult to analyze the behaviors of distributed enterprise applications. First, analysis can require detailed monitoring and evaluation, going beyond measurements of metrics like average CPU load, network bandwidth, number of tasks completed, etc. [17], but instead, desiring end-to-end information, meaning that analysis should capture the entire life-cycle of the request processing and cover every deployed resource in an enterprise, ranging from hardware to individual software components. Unfortunately, there currently exist no standard tools for capturing such detailed information. Second, while industry is developing standards for instrumenting distributed applications and systems [14], the XML-based representations used for the Common Base Event [25] monitoring standard and the application-level monitoring methods and interfaces provided by widely available tools like HP Openview [73] exhibit overheads that prevent their usage for capturing and analyzing detailed system- or application-level information about the precise resource usage associated with select application behaviors [75]. Third, the large diversity of applications routinely used in the enterprise and grid domains, ranging from simple *ftp* to complex distributed collaborations, makes it difficult to assume the existence of common, clean interfaces for performance evaluation. Finally, analysis cannot assume source code availability, since sources are not likely readily available for all of the applications being evaluated and managed by an organization.

One approach is to integrate generic methods for analyzing program performance into middleware, used in systems like Photon [89], Pinpoint [30], and many others [92]. The idea is to automatically observe a program's usage of middleware functions, including the middleware-mediated interactions between different, distributed application components. Applications need not be modified, and access to source code is not necessary. However, since actual resource usage is controlled by the operating system, it is not possible to accurately account for the performance effects of certain application- or middleware-level behaviors. A well-known example is the difficulty of mapping remote requests initiated by middleware to actual communication actions, due to TCP message buffering and protocol congestion behavior [47]. Basic causes of these problems include (i) system-level asynchrony, i.e., the OS kernel's internal use of concurrency to satisfy multiple application requests, and

(ii) system-level independence, i.e., the fact that OS kernels independently manage and allocate system resources for the multiple application-level processes being run. From the middleware level, therefore, it is difficult to attribute the usage of certain system resources to specific user-domain actions.

1.2 Goals

The focus of this thesis is on distributed applications with defined quality needs, as exemplified by the operational information systems used in large corporations [71], by IT infrastructures used in 24/7 settings, and online scientific enterprises, an example of the latter being the online monitoring, through runtime data visualization, for scientific simulations or experiments [96]. For such applications, our first goal is to understand, at runtime, their execution behavior relevant to the SLA-governed delivery of end user services. Key requirements are to do so *dynamically*, while applications are running and without predetermination in what behaviors to diagnose where, and *non-intrusively*, that is, without requiring extensive and difficult to maintain application instrumentation. Another requirement is *generality*, referring to the fact that runtime methods for behavior understanding should be applicable to a range of operational applications, rather than being limited by the need to use certain middleware or specific application programming environments.

1.3 Terminology

This section enumerates some of the terms and notions used in this thesis.

- **End-to-End:** in this thesis, end-to-end refers to the entire life-cycle of the request processing, and covers every involved resource in an enterprise, ranging from hardware to individual software components.

- **Service Level Agreement (SLA):** a contract between the client and the service provider detailing the services and the obligations associated with it [52]. A related term is **"Service Level Objective" or (SLO)**, which refers to the level of service to be delivered by the provider.

- **Request Class:** a request class denotes a set of requests with similar resource requirements. These classes can be defined in a variety of ways. The most common way to define a class is through "Service Level Agreements" or (SLAs). In this thesis, we will study different SLA classes and introduce some other class definitions appropriate for our analysis.

1.4 Thesis Statement

Scalable and dynamic system support for end-to-end analysis without application or end-user support can aid in understanding the behavior of enterprise systems and reduce the complexity of managing them.

1.5 Solution Approach: Service Paths – Scalable Analysis of Enterprise Applications

This thesis approach to scalable analysis of enterprise applications borrows terminology and methods from both the domains of performance monitoring and compilers. We capture end-to-end service delivery with a *service path*, which describes the sequence of services used by a request posed by an end user. This sequence is detected at runtime, using non-intrusive (i.e., not requiring application instrumentation) and dynamic techniques (i.e., dynamically inserted instrumentation). Given this sequence, a *critical service path* is the service path subset (i.e., a certain set of software components) involved in request execution that is relevant to the request's performance behavior. This differs from the *uses* relations defined in Software Engineering in that even a logging service can influence a request's performance behavior if such a service is performed synchronously with the remainder of the service path. The outcome is the ability to capture application-relevant performance metrics, such as end-to-end latency for remote requests, as well as understand the contributions to such latencies from computational components or network devices. The analyses performed are stochastic in nature, to curtail instrumentation overheads and more importantly, to reduce the level of knowledge needed about applications when analyzing them. Figure 1 shows the general architecture of service path instrumentation and analysis.

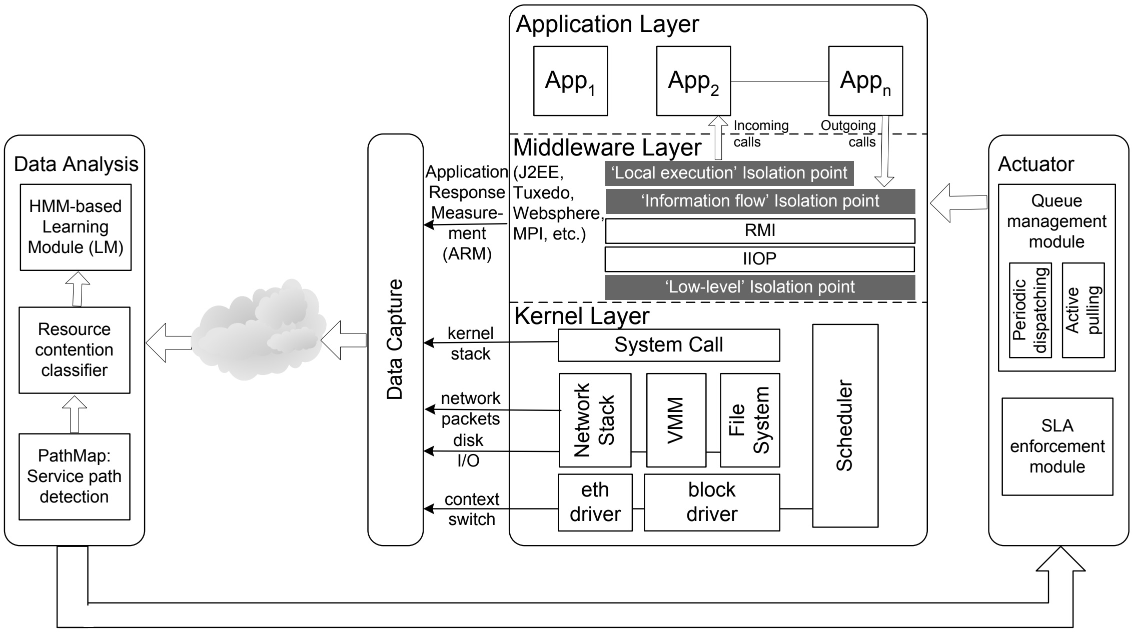


Figure 1: The Service Path Architecture

In general, it is important to first classify requests and then consider the sets of service paths (i.e., service graphs) traversed by different request classes. Such classifications might be provided by systems analysts (e.g. SLA class), or they might be determined automatically using techniques like Hidden Markov Models, for example [62]. The service paths taken by requests are dynamic and probabilistic, because they may depend on request types, parameters (and parameter values), and on the scheduling methods used by underlying middleware or systems, thus the need for their dynamic discovery.

1.6 Technical Contribution

The key technical contributions of this thesis are its conception, development, and implementation of the notion of dynamic service paths in enterprise applications. In addition, the thesis makes three concrete technical contributions:

- It develops a toolkit for online, end-to-end performance diagnosis of distributed systems, called **E2EProf** [1]. The toolkit uses a modified form of time-series analysis (commonly used in Digital Signal Processing or DSP), to detect the service paths taken by requests across a distributed IT infrastructure and delays incurred due to different path components. Since the toolkit does not require applications to be modified, it can also handle legacy components.

- A second contribution is a system-level monitoring toolkit called ***SysProf*** [8], which captures monitoring information at different levels of granularity, ranging from tracking the system-level activities triggered by a single system call, to capturing the client-server interactions associated with a service paths, to characterizing the server resources consumed by sets of clients or client behaviors. SysProf operates enterprise-wide in that it can track the application- and system-level activities and associated resource usage across the multiple machines used in typical enterprise infrastructures.

- The third contribution of the thesis is a publish-subscribe based monitoring data delivery framework called **QMON** [2]. QMON offers high levels of predictability for service delivery and supports utility-aware monitoring while also able to differentiate between different levels of service for monitoring, corresponding to the different classes of SLAs maintained for applications.

The idea is to provide a set of tools that make it easier to analyze the entire end-to-end service paths by discovering the dependencies across different service components using E2EProf; measuring the resources consumed while providing those service using SysProf; and delivering the monitored data using QoS-aware QMON channels.

1.6.1 E2EProf: Black-Box End-to-End Performance Understanding

Existing solutions to runtime performance understanding typically rely on code or application instrumentation (e.g., using ARM instrumentation standards [14]). This requires access to and manipulation of application sources or binaries, which may involve application experts and/or sophisticated analysis or instrumentation systems. Other solutions rely on offline analysis and/or do not scale to dynamic datacenter-level applications, particularly when such applications are composed of custom mixes of in-house, contracted, and third party codes, as is common in modern enterprise settings.

The *E2EProf* toolkit enables the efficient and non-intrusive capture and analysis of end-to-end program behavior for complex enterprise applications. E2EProf permits an enterprise to recognize and analyze performance problems when they occur – online, to take corrective actions as soon as possible and wherever necessary along the paths currently taken by user requests – end-to-end, and to do so without the need to instrument applications – non-intrusively. Online analysis exploits a novel signal analysis algorithm, termed *pathmap*, which dynamically detects the causal paths taken by client requests through application and backend servers and annotates these paths with end-to-end latencies and with the contributions to these latencies from different path components. Thus, with pathmap, it is possible to dynamically identify the bottlenecks present in selected servers or services and to detect the abnormal or unusual performance behaviors indicative of potential problems or overloads.

By leveraging kernel-level network packet tracing, pathmap can operate without the need for application instrumentation. By optimizing pathmap's computations and using compact trace representation, the space and time complexity of the algorithm are reduced to become orders of magnitude better than similar techniques previously described in the literature.

Pathmap and the E2EProf toolkit successfully detect causal request paths and associated performance bottlenecks in the RUBiS ebay-like multi-tier web application and in one of the datacenter of our industry partner, Delta Air Lines.

1.6.2 SysProf: Online Fine-grain System Monitoring

Runtime monitoring is key to the effective management of enterprise and high performance applications. To deal with the complex behaviors of today's multi-tier applications running across shared platforms, such monitoring must meet three criteria: (1) fine granularity, including being able to track the resource usage of specific application behaviors like individual client-server interactions, (2) real-time response, referring to the monitoring system's ability to both capture and analyze currently needed monitoring information with the delays required for online management, and (3) enterprise-wide operation, which means that the monitoring information captured and analyzed must span across the entire software stack and set of machines involved in request generation, request forwarding, service provision, and return.

The *SysProf* system-level monitoring toolkit provides a flexible, low overhead framework for enterprise-wide monitoring. The toolkit permits the capture of monitoring information at different levels of granularity, ranging from tracking the system-level activities triggered by a single system call, to capturing the client-server interactions associated with certain request classes, to characterizing the server resources consumed by sets of clients or client behaviors. SysProf's ability to carry out fine-grain monitoring is derived from two facts: (1) its kernel-level implementation, with direct access to the system-level resources involved in request execution, coupled with (2) its runtime configurability, making it possible to capture precisely the information needed and analyses required for current diagnosis tasks. Configurability also contributes to SysProf's effective role in runtime system management, enabling dynamic tradeoffs in monitoring granularity vs. overheads and delays. SysProf operates enterprise-wide in that it can track the application- and system-level activities and associated resource usage across the multiple machines used in typical enterprise infrastructures.

The thesis demonstrates the efficacy of SysProf by using it to manage two different enterprise applications: (1) detecting performance bottlenecks in a high performance shared network file service, and (2) enforcing service level agreements in a multi-tier auctioning web site.

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

QMON extends E2EProf and SysProf by incorporating Quality of Service (QoS) features in them. 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.

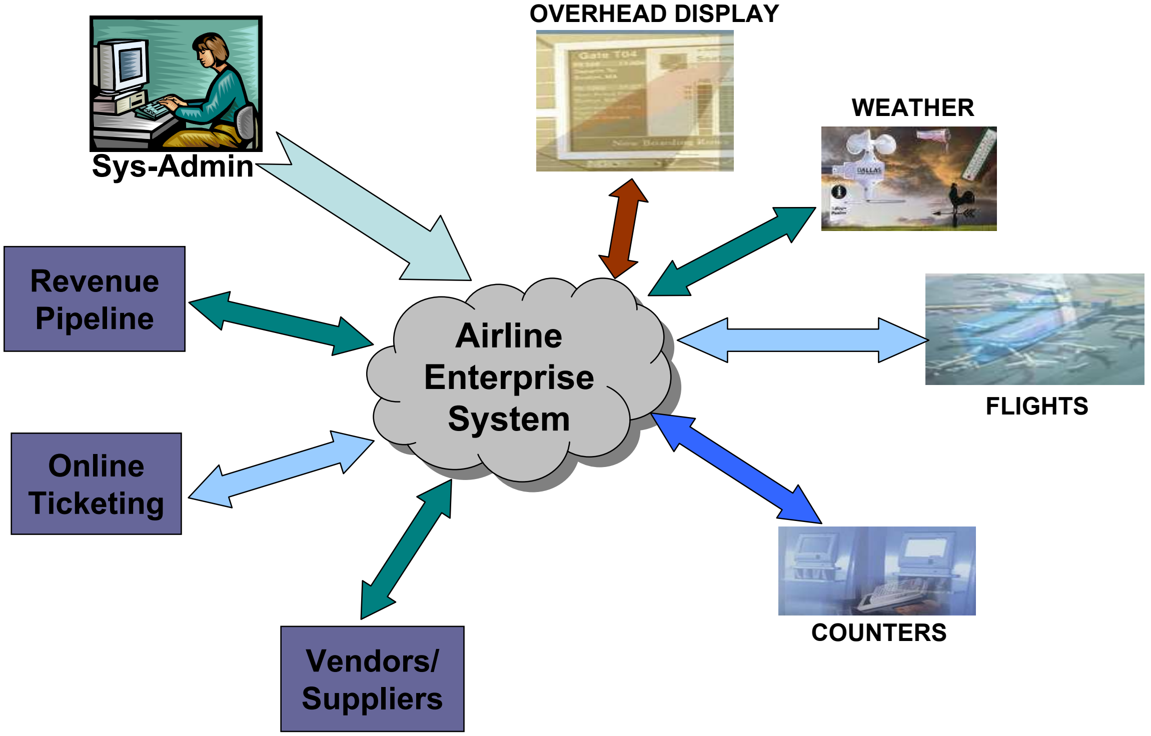


Figure 2: Operational Information System used by Delta Airlines

QMON supports utility-aware monitoring while also able to differentiate between different classes of monitoring, corresponding to classes of SLAs. That is, for each different class of service, QMON guarantees the delivery of monitored data as per the QoS required by that class. 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.

1.7 Representative Enterprise Applications

Service path analysis permits it to capture and deal with the dynamic behaviors of complex enterprise applications. A specific target class of applications addressed by this thesis are the *Operational Information Systems*(OIS) [43] used by large organizations for controlling day-to-day operations, an example being the OIS run by one of our industrial partners, Delta Air Lines, which provides the company with up-to-date information about all its operations

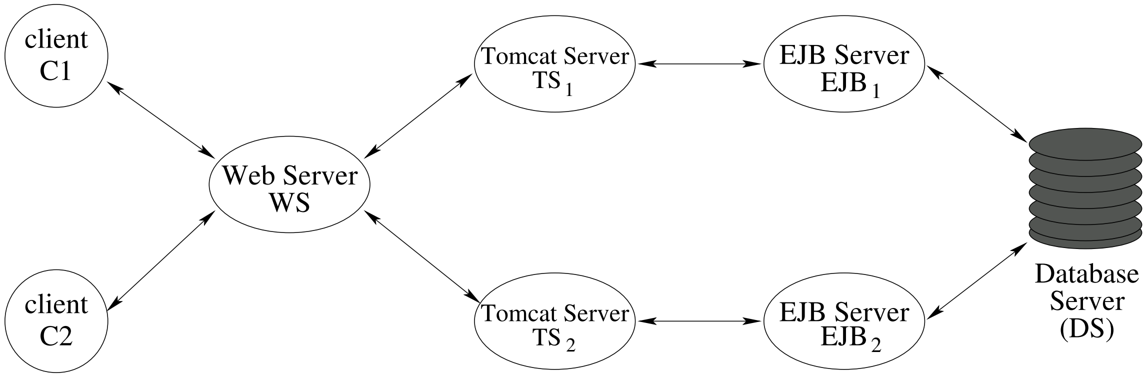


Figure 3: Multi-tier RUBiS application

like flights, crews, baggages, catering, etc (Figure 2.) Data is generated at multiple sources and consumed at different locations after applying one or more high level business logic. In order to function properly, these systems must operate and adapt to changes within well-defined constraints derived from their Service Level Agreements (SLAs) and dependent on the business values or utilities associated with their various services [95]. If a SLA is violated, system administrators usually analyze large complex logs in order to isolate faulty components. Service path analysis can be used to automate performance diagnosis, thereby reducing such maintenance costs. In section 2.3.3, we will describe a subsystem of Delta's OIS called "*Revenue Pipeline System*" in detail and show how to extract dependency and latency information from its traces without making any application-specific assumptions.

Another enterprise-scale multi-tier application that we studied extensively is called RUBiS (Figure 3.) RUBiS is an open source multi-tier online auction benchmark developed by researchers at Rice University [26]. RUBiS implements the core functionalities of an auction site like selling, browsing, and bidding. RUBiS is available in three different flavors: PHP, Java HTTP Servlets and Enterprise Java Beans (EJB). We evaluate the overhead and accuracy of E2EProf and SysProf, and demonstrate how these tools can be used for online performance debugging in these applications.

In addition to RUBiS and Delta's OIS, this thesis also studies various performance issues in shared NFS service like the one used in Virtual storage architecture [76, 5, 11]. The back-end storage servers are hidden from the client's view by a user-level proxy that interposes every request from the client to the server. A typical problem in these environments is to detect failures and performance bottlenecks. One of the ways to detect this is by tracking the execution of the requests through different components and measuring the latencies and resources consumed. However, this is a difficult problem because there may be a large number of nodes involved through which the requests pass and get processed. In section 3.4.2, we illustrate how SysProf can be used to detect performance bottleneck in a virtual storage service.

1.8 Organization

The remainder of this dissertation is organized as follows:

Chapter 2 presents the service path in detail. It lists down the assumptions we make in deriving the end-to-end relationships and describes various components of E2Eprof.

Chapter 3 describes our SysProf architecture and the algorithms it uses to do micro-level performance analysis.

Chapter 4 describes our publisher-subscriber QoS- and utility-aware monitoring information dissemination framework called QMON.

Chapter 5 reviews related work on end-to-end analysis and compare them with our service path approach.

Chapter 6 concludes with a discussion on the most important contribution of this thesis, the lessons learnt and suggestions for future work.

CHAPTER II

E2EPROF: BLACK-BOX END-TO-END PERFORMANCE DIAGNOSIS

2.1 Introduction

The processing of a single request in an enterprise system can generate intricate interactions between different components across many machines, making it hard even for experts to understand system behaviors. A concrete example are the '*poison messages*' experienced in the IT infrastructure run by one of our industry partners [61]. This chapter introduces an online, end-to-end toolkit called 'E2EProf' that discovers these interactions without modifying any application components and without making any application specific assumptions. E2EProf can be used to diagnose the performance problems that arise from complex interactions across multiple subsystems and machines. First, its methods for *end-to-end* performance understanding can capture the entire life-cycles of requests as they are being processed by an enterprise application's many hardware and software components. Second, E2EProf analysis enables *online* problem diagnosis, because of its optimizations and compact trace representations. In comparison, ongoing industry developments are seeking generality and interoperability, through the ARM standards for instrumenting distributed applications and systems [14], the XML-based representations used for Common Base Events [25], and the general symptom databases and rule engines used in widely available tools (e.g., HP's Openview [73] or IBM's Tivoli [88]). The overheads exhibited by these industry-provided tools prevent their usage for capturing and analyzing the detailed system- or application-level information required for online, end-to-end problem diagnosis. In the critical IT infrastructure used by one of our industry partners, for instance, instrumentation cannot be turned on without compromising required performance, thereby making it impossible to use it for continuous problem detection and system management [53]. Third, since E2EProf uses *non-intrusive* kernel-level network tracing for application monitoring,

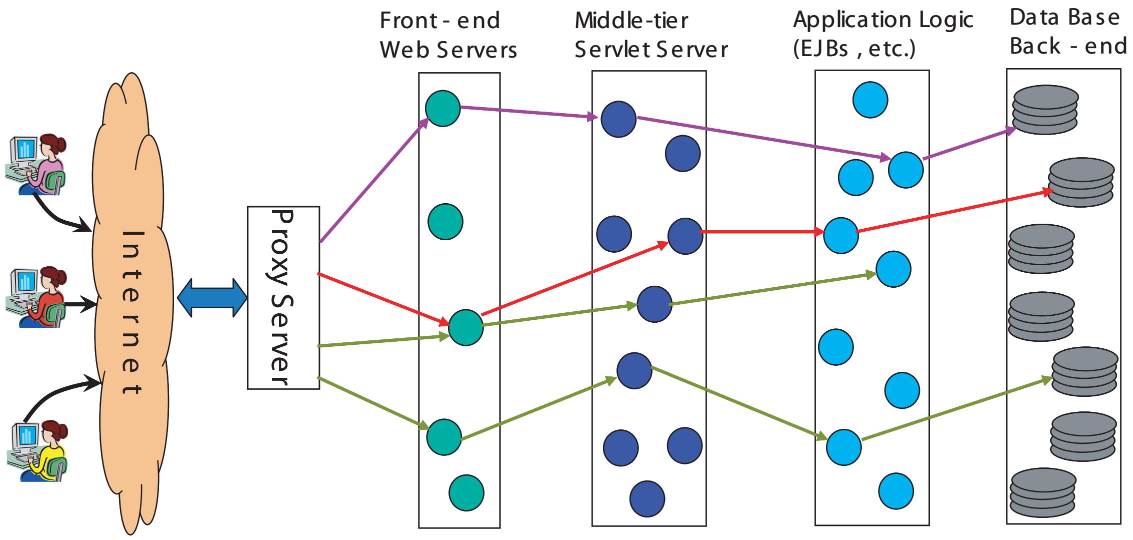


Figure 4: Example ServicePath in a multi-tier web service

it can operate across the large diversity of applications routinely used in the enterprise domain, without the need to assume the existence of common, clean, and perhaps most importantly, without requiring uptodate monitoring instrumentation. Fourth, E2EProf operates without requiring access to source code, since it is not likely readily available for all of the applications being evaluated and managed by an organization. In fact, even if sources were accessible, the lack of proper documentation often makes it a daunting task to analyze these extensive codes.

The *E2EProf* toolkit uses an incremental approach to performing *end-to-end* analyses of request behaviors. Specifically, it encapsulates different request interactions across distributed program components with different '*service paths*', where each such path describes a set of dynamic dependencies across distributed components formed because of the services they provide and the requests they service. Figure 4 shows the multiple service paths used by three different types of clients in a multi-tier web service, for example, where paths are differentiated by the kinds of requests being submitted.

Online service path encapsulation is *non-intrusive*, that is, it does not require modifying program components. This is done by correlating the timestamps of the messages exchanged between interacting components. E2EProf's cross-correlation analyses can capture application-relevant performance metrics, such as the end-to-end latencies experienced by requests, and they can determine the contributions of specific application-level services and network communications to such latencies. Thus, E2EProf does not require changes to user-level code, including changes that would recompile it (e.g., with a debug switch). Further, the choice of requests, components, and service paths to be analyzed can be changed at any time, without the need to recompile, re-link, or re-edit programs.

While the idea of path-based analysis been used by other researchers to discover faults and performance problems in distributed systems [9, 30, 19, 87, 81], E2EProf makes the following unique contributions:

- Its *service path* abstraction can be used to encapsulate the causal paths of different requests (or services), capture end-to-end request delays, and the components of those delays due to each individual software component.
- Its time-series analysis algorithm, termed *pathmap*, discovers the causal request paths from network packet traces non-intrusively, which means pathmap neither requires access to application source code, nor modifications to deployed application services.
- Its ability to understand the performance of complex distributed applications is demonstrated by carrying out detailed online performance analyses for the RUBiS multi-tier auctioning web application.
- The low latency, efficient analyses performed by E2EProf permit it to be used for online management, using a black-box scheduling algorithm to manage Service Level Agreements (SLA) in RUBiS.
- E2EProf has gone beyond in-lab concept demonstrations, by using its pathmap algorithm to evaluate the performance of an enterprise application deployed in one of our industry partner's datacenters, the 'Revenue Pipeline' subsystem used in Delta Air Line's Atlanta datacenter.

E2EProf is the outcome of a multi-year effort to develop efficient mechanisms and methods for runtime performance understanding. Its kernel-level monitoring techniques build on our earlier work on the Dproc methods for monitoring cluster or blade servers [6, 7]. Insights about the need to not only monitor with low overheads or perturbation, but also with defined monitor with low overheads or perturbation [45], but also with defined quality characteristics, stem from joint work with researchers at HP Labs [2]. E2EProf shares its property of being able to run with unmodified application codes and without access to source codes with systems that integrate monitoring capabilities into middleware, including Photon [89], Pinpoint [30], and others. many others [92]. These systems can automatically observe a program's usage of middleware functions, including the middleware-mediated interactions between different, distributed application components. E2EProf complements such middleware-level monitoring, because lacking such methods' intimate knowledge of middleware function and implementation, it cannot directly attribute program (mis)behavior to certain middleware functions [61].

In the remainder of this chapter, we describe the service path abstraction and various components of E2EProf toolkit. The next section describes the pathmap algorithm and analyzes it in detail. Experimental evaluation is presented in Section 2.3 together with some realistic test cases of performance diagnosis and management. Section 2.4 surveys the related work. Summary appears in Section 2.5.

2.2 End-to-End Service Path Discovery

2.2.1 Basic Abstractions, Methods, and Assumptions

Modern enterprise systems (e.g., multi-tier web services) are composed of multiple hardware and software components that dynamically interact to provide services to clients. Furthermore, different client requests may belong different client requests may belong to one or more *service class(es)*, which are defined on the basis of simple request types, clients IDs, or more generally, *Service Level Agreements* SLAs. These requests may take different paths through the enterprise software, invoking different and multiple software components before responses are generated. We term the ensemble of paths taken by client requests in different service classes as '*Service Paths*'.

Service paths form the basis of E2EProf's online end-to-end performance analyses, because they characterize the end-to-end properties sought by the enterprise and capture the complex dependencies that exist across the different software components involved in service provision. For each path, E2EProf's analyses can describe not only the path's end-to-end latency but also the latencies incurred across different path edges, which can be used to pinpoint the bottleneck components in a request path. Therefore, service path analysis can pinpoint the bottleneck components in a request path, and it can be used for provisioning, capacity planning, enforcing SLAs, performance prediction, etc.

The first step in service path analysis is to discover the service paths of different service classes. Toward this end, the *pathmap* algorithm uses a time-series analysis of the network packets flowing through the enterprise network. The technique was first proposed by Aguilera *et al.* [9] for black-box performance debugging, and their '*convolution algorithm*' uses time-series analysis to establish causal relationships across different components by cross-correlating message traces collected from different network edges. Their algorithm, however, is computationally expensive, primarily targeting offline analysis. In comparison, this paper proposes a series of optimizations that jointly enable service path discovery in real time. The following assumptions are made by the *pathmap* algorithm:

- Each client's requests belong to a unique *service class*, which is known to the front end (i.e., the first nodes in the distributed system that receives the request). A service class is defined *a priori* based on the type of client or on the type of request. In current enterprise settings, such definitions use Service Level Agreements (SLAs), each of which is a contract between the client and the service provider, detailing the desired services and the obligations associated with their use [52]. Pathmap assumes that requests belonging to the same service class have similar resource requirements.

- A request path can either be unidirectional (as in streaming media applications) or bidirectional as in the request-response conduits used in multi-tier web services. In the latter case, responses traverse the same set of nodes as the corresponding requests, but in reverse order.

- Pathmap assumes that the distributed application and system are operating in steady state during the analysis 'time window', where deviations are due to internal anomalies or external drastic changes in system usage. Such anomalies occur when a node malfunctions, when a network link goes down, or when a buggy application overloads the system, for example. A sample abnormal external change may be a malicious attack or a sudden increase in user interaction (e.g., the *Slashdot effect.*)

- At small time scales, there may be large variability in the processing of individual requests, but in steady state, the system is assumed to be adequately provisioned so that the queuing and processing delays at each of its nodes don't significantly change the distribution of the intermediate responses (generated as a result of partial processing of the requests at the intermediate nodes in the path), as compared to the arrival distribution at the front-end. Pathmap can, however, accommodate changes in rate across nodes (e.g., an EJB server issuing multiple data base queries for a single client requests).

  Stated intuitively, different kinds of traffic (i.e., different service classes) will likely have different queuing/processing delays, more so than requests within each single class. This fact permits us to distinguish between them. This makes sense because the factors that affect delays include request content (or type of content, e.g., dynamic/static,media/text) and class-based network and system policies (e.g., priorities, load distribution strategies and maintenance schedules.) Within this context, we do take into account, however, certain symmetrical situations, such as those arising from round-robin load balancing across multiple nodes that serve just one type of traffic.

2.2.2 System Representation

Formally, a distributed application or system may be described as a directed graph G(V,E), where the vertices in the graph represent application components and the edges represent their logical communication links. The *service graphs* considered in this thesis are comprised of nodes that may be processes, threads, or machines, communicating with each other via network links. [1]. Thus, in the service graphs considered in E2EProf's evaluation, all

---

[1] Although we consider only network communication links, the E2EProf approach can also be extended to IPC mechanisms like *pipes* and *message queues*.

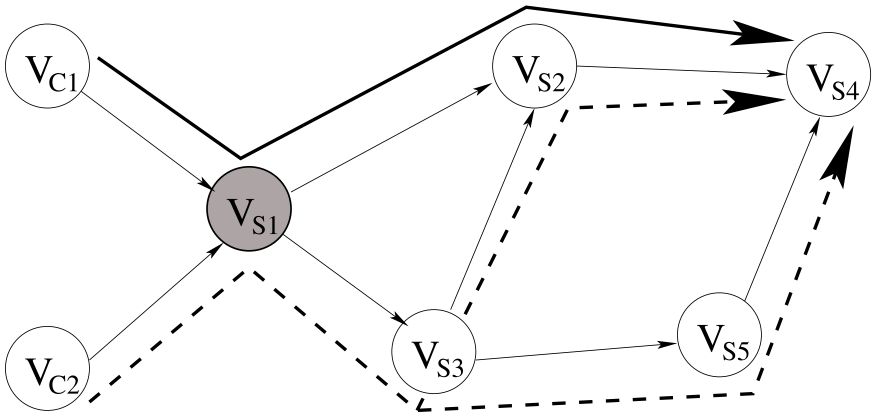


Figure 5: Example Service Graph: V_{c1} and V_{c2} are the client nodes and V_{sn} are service nodes.

components communicate by exchanging network packets.

Each service graph has two type of nodes: client nodes(V_C) and service nodes (V_S). Similarly, the edges are of two types: *client edges* (E_C,one of the vertices is a client node) and *service edges* (E_S, both vertices are service nodes.) Requests originate in client nodes, where we assume that the requests issued by each particular client node belong to the same service class. A physical client issues multiple classes of requests will be modelled as multiple client nodes, one per request class. Service nodes house software components that operate on requests. They are labelled by their IP addresses or by a combination of their IP addresses and process IDs, depending on whether there is one or more service node per physical machine node (e.g., an application server and database server being located on the same physical machine).

Edges denote logical communication link between service nodes. These logical connections are characterized by source and destination address pairs. They may be transient, which will usually be the case in front-end servers, or persistent, which is typical for middle and back-end servers. Furthermore, a single connection may consist of aggregated traffic from separate clients, and it may therefore, exhibit multiple traffic patterns. Figure 5 depicts a sample service graph. For this graph, the goal of the *pathmap* algorithm is to compute the paths of requests from each client node through the service graph, along with the delays incurred in traversing the edges and nodes in those paths.

2.2.3 Pathmap Algorithm

The pathmap algorithm is designed to meet the following criteria:

- *Non-intrusive*: traces are collected without the need to modify program components, relink them, and/or restart applications.
- *Adjustable perturbation*: to keep tracing overhead within '*acceptable*' limits, levels of detail and precision in tracing can be varied to meet application needs.
- *Always on* – online – tracing: trace data is a never-ending stream of events originating from continuously monitored service nodes.
- *Dynamic analysis*: trace capture is combined with online analysis over some '*window*' of trace data, continuously updating 'old' models of program behavior.
- *Finite history*: online analysis does not revisit past *windows* over trace data streams.

The pathmap algorithm relies on the E2EProf tracing subsystem, which uses standard operating system facilities to collect timestamp traces for every (*source*, *destination*) pair of inter-component messages at each service node. Traces are not collected from client nodes, since those are usually beyond the reach of enterprises. Traces contain the source and destination IP of network IP packets and the times at which they enter or leave service node interfaces. The key idea of the pathmap algorithm is to convert these traces to per-edge time series signals and then compute the cross-correlations of these signals. Specifically, if a signal f contains a copy of the signal g, then their cross-correlation signal $(f \star g)$ has a distinguishable spike at position d, where d is equal to the time that the copy of g in f has shifted from g. This kind of correlation analysis is commonly used in digital signal processing to compute the level of similarity between two signals.

First introduced by Aguilera *et al.* [9] in a similar context, pathmap uses cross-correlation analysis to discover the most probable request paths in a distributed system. Consider the request path $(V_{C1} \rightarrow V_{S1} \rightarrow V_{S2} \rightarrow V_{S4})$ shown in Figure 5. Let $T^{x}_{x \rightarrow y}$ be the time series signal of the messages from x to y collected at the node x, and $T^{y}_{x \rightarrow y}$ be the time series signal for the same set of messages collected at node y. The cross-correlation plot of $T^{s1}_{c1 \rightarrow s1}$

and $T^{s1}_{s1\rightarrow s2}$ (denoted by $corr(T^{s1}_{c1\rightarrow s1}, T^{s1}_{s1\rightarrow s2})$) has a spike at position d, where d is the time that V_{s1} takes to process V_{c1}'s request. This implies that there is a causal relationship between messages on edge $V_{c1} \rightarrow V_{s1}$ and messages on edge $V_{s1} \rightarrow V_{s2}$. Similarly, the cross correlation plot $corr(T^{s1}_{c1\rightarrow s1}, T^{s2}_{s2\rightarrow s4})$ also has a spike, and its position is the sum of the communication latencies at the two edges ($V_{C1} \rightarrow V_{S1}$ and $V_{S1} \rightarrow V_{S2}$) and of the computation latencies at the two vertices (V_{s1} and V_{s2}). The presence of the spike also indicates a causal relationship between messages on edge $V_{c1} \rightarrow V_{s1}$ and messages on edge $V_{s2} \rightarrow V_{s4}$. The cross-correlation plot $corr(T^{s1}_{c1\rightarrow s1}, T^{s1}_{s1\rightarrow s3})$, however, has no distinguishable spike as no requests from V_{c1} pass through V_{s3}.

Algorithm 1 Pathmap

```
  Let $W$ = Length of sliding window
  Let $\Delta W$ = Service Graph refresh interval
  Input: Online time series data streams from service nodes

  function ServiceRoot()
  for all Service node $S_i$ that are at the front-end do
    for all Client nodes $V_c$ connected to $S_i$ do
      Service Graph $G_c = \{\}$
      Add $S_i$ in Graph $G_c$
      Add an edge $E_c(V_c \rightarrow S_i)$
      ComputePath($G_c$, $T^{S_i}_{V_c\rightarrow S_i}$, $S_i$)
    end for
  end for

  function ComputePath($G_c$, $T_c$, $S_i$)
  Mark $S_i$ as visited
  Let $S_d$ = List of destination nodes $S_i$ is connected to
  for all $d_s$ in $S_d$ do
    $corr$ = ComputeCrossCorrelation($T_c$, $T^{d_s}_{S_i\rightarrow d_s}$)
    $P$ = List of spike's position in $corr$
    if $P$ is not empty then
      if vertex $d_s$ not in $G_c$ then
        Add vertex $d_s$ in $G_c$
      end if
      Add an edge $E_s(S_i \rightarrow d_s)$ and label it with $P$
      if $d_s$ not visited then
        ComputePath($G_c$, $T_c$, $d_s$)
      end if
    end if
  end for
```

The above example illustrates how correlation can be used to establish causality between client edges and service edges. Given this background, Algorithm 1, outlines the actual pathmap algorithm. It takes as input the time-series data streams computed from the message timestamps collected at different service nodes. The most recent *sliding window* of size W is maintained for each of these streams. After every time interval ΔW, the '*ServiceRoot*' function is invoked to update the service graphs for all *clients* belonging to different service classes. Thus, instead of analysing the whole time series, *ServiceRoot* maintains a *sliding window* of size W and computes service graphs based on the most recent window. For the analysis to be statistically significant, the size of W is chosen such that it contains large number of requests. The algorithm starts tracking the path at the front-end service nodes, which become the roots of service graphs. In addition, it adds an edge between the client node and the root vertex and then calls *ComputePath* to calculate rest of the graph.

ComputePath is the heart of the pathmap algorithm. Its parameters are a partial service graph G_c, a time-series signal (T_c) of the incoming requests of the service class (say C) at the front-end for which the service graph G_c is being determined, and the service node (S_i) to be processed next. ComputePath finds the next set of service nodes used by the request class represented by the time-series T_c. This is done by the process of correlation described above. Basically, T_c is cross-correlated with the time-series signal of the message (or network packet) traces collected from the nodes(d_s) adjacent to S_i. If the correlation is high (as indicated by the presence of the spikes), then there exists a path from S_i to d_s taken by the requests belonging to service class C. This is recorded by adding vertex d_s into the graph G_c (if such a vertex does not yet exist) and by adding an edge from S_i to d_s. The edge is labelled with the *delay*(s) as denoted by the spikes' position in the cross-correlation test. This delay is the sum of the time taken by the request to arrive at node S_i, the processing delay at node S_i, and the communication delay in the path from S_i to d_s. The computing delay at node S_i is the difference of the delays corresponding to its incoming and outgoing edges. The existence of more than one spike indicates that the request may have taken different paths to S_i (e.g., $S_1 \rightarrow S_2 \rightarrow S_i \rightarrow S_4$ and $S_1 \rightarrow S_3 \rightarrow S_i \rightarrow S_4$). Once the path to d_s is established, the algorithm proceeds further by performing a recursive depth-first search and exploring other edges in the service graph.

Spikes in the cross-correlation series are detected by finding *points* that are local maximas and exceed a threshold ($mean + 3 \times Std.Dev.$). In traces with some noise, or when the parameters τ and ω are set to very small values, there may exist spikes that are very close to each other. To address this issue, we define a resolution threshold window ($5 \times \tau$) that chooses only the tallest spike in a particular window.

Note that the algorithm uses the time-series signal ($T^{d_s}_{S_i \to d_s}$) at S_i's neighbours (d_s) to compute the cross-correlation instead of the time-series signal ($T^{S_i}_{S_i \to d_s}$) at S_i itself. This is because the former does not include the communication delay incurred in the edge $S_i \to d_s$. It is possible to separately calculate these two cross-correlations and determine communication delay from their difference. Nevertheless, both time-series signal ($T^{d_s}_{S_i \to d_s}$ and $T^{S_i}_{S_i \to d_s}$) are sufficient to determine the existence of path between S_i and d_s.

2.2.4 Computing Cross-Correlation

The most expensive step in the pathmap algorithm is computing the cross-correlation. The basic formulation of the discrete cross-correlation shown in Eq. 1 can be computed in $O(n^2)$ time.

$$Corr_d(x,y) = \frac{\sum_{i=0}^{n-1} (x_i - \overline{x})(y_{(i+d)} - \overline{y})}{\sqrt{\sum_{i=0}^{n-1} (x_i - \overline{x})^2}\sqrt{\sum_{i=0}^{n-1} (y_{(i+d)} - \overline{y})^2}} \qquad (1)$$
$$where, \quad \text{d = 0,1,...,(n-2),(n-1)}$$

The *cross-correlation theorem* (Eq. 2) provides an efficient alternative to compute cross-correlation. The *Fourier transform* can be computed using FFT(*Fast Fourier Transform*), which reduces the time to calculate cross-correlation from $O(n^2)$ to $O(n \log n)$.

$$x \star y = \mathcal{F}^{-1}\left[\mathcal{F}[x]\mathcal{F}[y]^*\right] \qquad (2)$$
$$where, \quad \mathcal{F} \text{ denotes Fourier transform, and}$$
$$z^* \text{ denotes the complex conjugate of z.}$$

Although FFT-based computation is more efficient and is the *de facto* standard in computing the cross-correlation of two arbitrary signals, Eqn. 2 computes cross-correlation for the full range of delay corresponding to the input time series. That is, if the length of the *sliding window* is 10 minutes, the length of the cross-correlation series is also 10 minutes. However, there are scenarios when correlation needs to be evaluated for short delays only.

For our analysis, we choose the direct cross-correlation method (Eqn. 1), because it can be adapted easily for incremental computation of correlation metrics, in addition to other optimizations. These significantly reduce its computational overheads, thereby making online service path discovery possible. The first optimization is based on the fact that most transactions in a distributed system are just a small fraction of the *sliding window*. Since our goal is to find the service transaction delays and not the full range of cross-correlation series, by assuming an upper bound (say T_u) on the transaction delay, the time complexity of computing cross-correlation directly (i.e., without FFT) between two time-series of duration W is drastically reduced from $O\left([\frac{W}{\tau}]^2\right)$ to $O\left(\frac{T_u}{\tau} \cdot \frac{W}{\tau}\right)$. τ is the time quanta or the smallest delay of interest. In comparison, the time complexity of FFT-based cross-correlation (Eqn. 2) is $O\left(\frac{W}{\tau} \log \frac{W}{\tau}\right)$, which is less than the $O\left(\frac{T_u}{\tau} \cdot \frac{W}{\tau}\right)$ even for small values of T_u. Fortunately, direct cross-correlation is incremental (as discussed earlier), and therefore, it can be computed over only the newly appended trace of size ΔW. This reduces the time complexity of direct cross-correlation further, to $O\left(\frac{T_u}{\tau} \cdot \frac{\Delta W}{\tau}\right)$.

A third important optimization is based on the fact that the network packet traffic in the Internet and in most enterprise systems is inherently bursty. This burstiness can be due to system or user behavior [33, 13], or it can be due to the lower level network protocol (e.g., TCP) behavior and network queuing [51]. In addition, a single transaction may be composed of multiple packets sent back-to-back. Bursty behavior results in dense network packet traffic intermixed with 'long' quiet zones. Our optimization takes advantage of this fact by simply omitting to compute correlation in the 'quiet' region, without compromising the accuracy of the result. This is done by computing the *time series* in such a way that the entries with value 0 (i.e., zero packets seen at the time corresponding to that entry) are discarded. As a result, the length of the *time series* trace is reduced by a large margin (more than 10 times for some of our enterprise traces). This not only decreases the computation time of the direct cross-correlation, (see Eqn. 1), but also increases the efficiency (both in

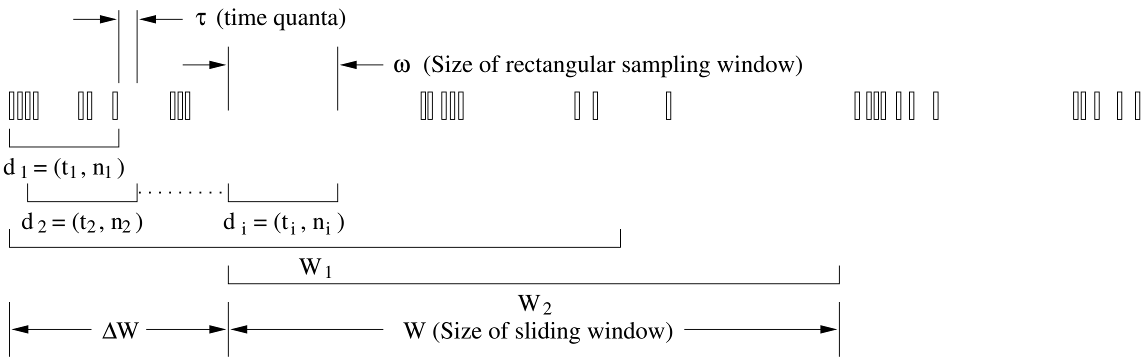


Figure 6: Time series computation

time and space) of collecting the trace at each service node, as we shall see in the next section. In summary, assuming that the average factor of *time series* reduction is 'k', the time complexity of direct cross-correlation drops to $O\left(\frac{T_u}{\tau} \cdot \frac{(\Delta W)/k}{\tau}\right)$. It should be noted that the reduction factor 'k' is dependent on the type of the distributed system and its workload, as well as the value of *time quanta* τ. A lower value of τ (i.e., finer grain analysis) results in a higher value of 'k'. This may increase the overall time required to compute cross-correlation because the decrease in τ may offset the effect of increase in 'k'.

2.2.5 Computing Time Series

The message traces collected at service nodes are converted to time-series data using a function $d(i)$, which represents the '*density*' of the packets at time instant $i \cdot \tau$ (or *ith* time quanta). A separate time-series is computed for every (*source*, *destination*) pair. $T^x_{x \to y}$ denotes the time series signal of the network packets from x to y collected at the node x. The density function estimation is based on two parameters: time quanta (τ) and the size of *rectangular sampling window* (ω), an integral multiple of τ.

$d^x_{x \to y}(i) =$ square root of number of messages at service node x transmitted
to y in time interval $\left[i \cdot \tau - \frac{\omega}{2},\ i \cdot \tau + \frac{\omega}{2}\right]$

Figure 6 shows a pictorial representation of time series computation. The message arrivals are shown as small rectangular boxes. Both W (size of sampling window) and ΔW (refresh interval) are also integer multiples of τ. Note the entry $d_i = (t_i, n_i)$ in the time-series computation in Figure 6. No packet was received during the *ith* sampling window, and therefore, as discussed in the previous section, d_i is not recorded in the time-series. The size of time quanta τ determines the resolution of the analysis. For a given sliding window size (W), a small τ results in longer time-series ($\frac{W}{\tau}$) and a proportional increase in the cost of servicepath analysis. Its value, therefore, should not be arbitrary small, but equal to the shortest service delay of interest. The purpose of the rectangular sampling window is to reduce the effect of variance in delay and suppress infrequent paths that occur due to the noise in the trace. A very small ω may produce many spikes during cross-correlation analysis resulting in false delays/paths. On the other hand, a large value of ω may overgeneralize the result (collapsing two spike into one, for example). For the systems we have analyzed, $\omega = 50 \cdot \tau$ gave the best set of results.

We note that although we have used a rectangular window for sampling, there are other windowing schemes (e.g., Hanning, Hamming) that are more robust to noise. The advantage of the rectangular window is that it is more efficient and requires constant time for incremental update (i.e., for calculating the next density function in the time-series). In other windowing schemes, updates require a computation time that is proportional to the size of the window. A more detailed discussion is beyond the scope of this thesis.

The process of time-series computation is further optimized using run-length encoding (RLE). Upon close examination of the time-series of actual enterprise traces, we found that there are many repeatable sequences, which provide substantial room for compression. RLE is particularly appropriate for this purpose, because it can be computed online, with negligible compression and decompression overheads. This not only reduces the network transmission overhead (when the time-series data is streamed to the remote node for analysis), but it also decreases the cost of cross-correlation analysis because the correlation of overlapping sequences in the series (Eqn 1) can be computed in a single step. The resultant time-series becomes a 3-tuple series (t, c, n) (one tuple for each *run*), where t is the timestamp of the first density function entry in the *run*, c is the length of the *run* and n is the value of density function (i.e., the number of packets in the sampling window (ω)). Additional evaluations of exact cost and savings are discussed in the experiment section (Sec. 2.3.)

2.2.6 Trace Collection

One of the requirements of service path analysis is that no application components should be modified or restarted. Also, the system should experience as little perturbation as possible. Our analysis requires timestamps and (source, destination) identification of the inter-component messages. These messages may be collected at various levels: at the application level (e.g., apache web server's access logs), at the middleware level (e.g., J2EE-level tracing [30]) or at the system and network level. The problem with tracing transactions at the application- or middleware-level is that there is not a single and widely deployed standard. *Application Response Measurement* (ARM) [14] is one such standard for monitoring transactions end-to-end in enterprise systems. The ARM standard was proposed in 1996 by a consortium of companies, but it still has limited acceptance. Other adhoc application or middleware instrumentations may require deep knowledge about application semantics as well as access to source code, which make it cumbersome for system administrators, who want to monitor their systems with as little effort as possible.

Passive network tracing provides a convenient way of listening to the interactions between different *service nodes*, without the need to modify any system components. Network packet traces may be collected from ethernet switch with *port mirroring* support or directly from service nodes by running *tcpdump* [2]. The traces obtained can be streamed to some central location for analysis. Although, this looks like a simple and attractive approach, it limits the scalability of our overall servicepath analysis. This is because the analysis node has to first compute the time series and then the service paths. Offloading the time-series computation to the service nodes decreases the work on central node. Also, the time-series can be calculated directly from the network activity at the service nodes instead of first logging the raw packet traces (using tcpdump) and then converting it to time-series signals. Towards this end, we implemented a linux kernel module called *tracer*, which uses the '*netfilter*' hooks to listen to the packets in the network stack Figure 7 shows the different steps in time-series and service path computation. The tracer module runs in each

---

[2] www.tcpdump.org

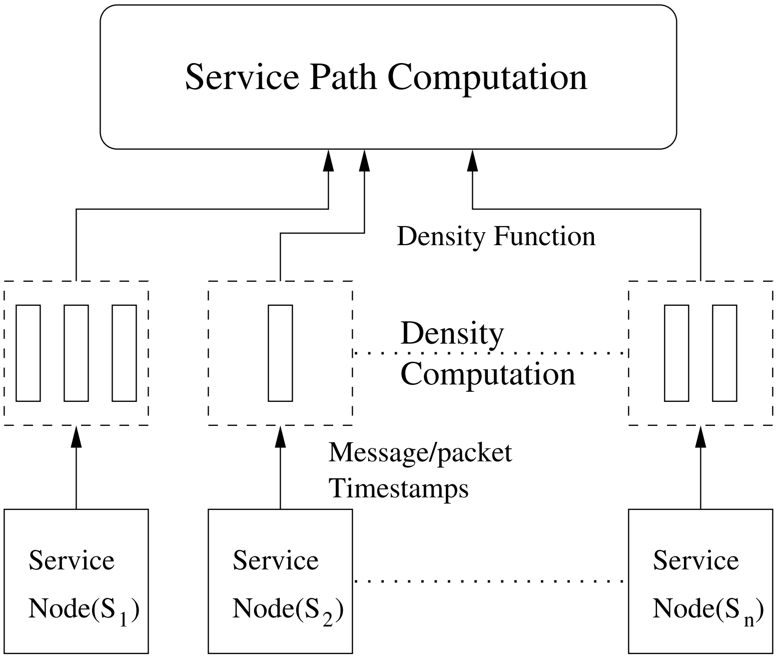


Figure 7: Service Path Computation Flowchart

service nodes and streams *REL*-encoded time series data. Each entry in the time-series is a 3-tuple $((t, c, n))$. There is one unique series for every (*source*, *destination*) IP pair seen at the service node where the tracer runs. The (source, destination) information is stored as meta-data in the time-series stream they represent. When a new packet enters or leaves a service node, the tracer increments the current *sampling window* and update the last tuple or create a new tuple in the time-series it belongs.

Our network packet *tracer* could have been implemented as an user-level application using the commonly available packet capture (*pcap*) libraries. The user-level solution is more portable but it incurs additional overhead from context switches and from the movement of trace data across the kernel-user space boundary. *Tracer* on the other hand, uses an efficient double buffering scheme to compute the time series directly and stream transfer it to the analyser node. Also, it provides flexibility to modify the values of τ and ω at run-time to change the resolution and/or overhead of the service path analysis.

2.2.7 Complexity Analysis

The overall time complexity of our pathmap algorithm using direct cross-correlation is $O\left(E \cdot [\frac{W}{\tau}]^2\right)$, where E is the total number of edges in the service graph, W is the sliding window size and τ is the *time quanta*. After applying all optimizations discussed in previous sub-sections, the time complexity is reduced to:

$$O\left(E \cdot \frac{T_u}{\tau} \cdot \frac{(\Delta W)/(k \cdot r)}{\tau}\right),$$

where T_u is the maximum possible transaction delay and ΔW is the service graph update interval. k is the optimization factor achieved by skipping quiet intervals in the packet traces and r is RLE compression factor. Assuming $W = m \cdot \Delta W$, the above can be rewritten as:

$$c_1 \cdot \left[\frac{1}{k \cdot r \cdot m} \cdot \frac{T_u}{\tau} \cdot E \cdot \frac{W}{\tau}\right],$$

where c_1 is a constant. On the other hand, the complexity of FFT-based cross-correlation (Eqn. 2) is $c_2 \cdot \left[E \cdot \frac{W}{\tau} \log \frac{W}{\tau}\right]$, where c_2 is a constant and is much larger than c_1. Comparing the two equations, it is easy to see that our optimized direct cross-correlation approach is much more time efficient than FFT-based computation.

The pathmap algorithm receives a total $2 \cdot E$ number of time-series signal streams from the service nodes, two from the two nodes connected by an edge. It stores the cross-correlation vectors (of size $\frac{T_u}{\tau}$) and a history of time-series (of the size of sliding window $\frac{W}{\tau}$) for each of these edges. The total space complexity, therefore, turns out to be $O\left(2 \cdot E \cdot (c' \cdot \frac{T_u}{\tau} + c'' \cdot \frac{W/(k \cdot r)}{\tau})\right)$.

It is worth mentioning here that the factors k and r mentioned above are dependent on the workloads and the distributed systems under analysis. The most extreme scenario is when both k and r are equal to one (their minimum possible value), in which case, FFT-based cross-correlation may perform better because the logarithmic term of $\log \frac{W}{\tau}$ is much smaller than $\frac{T_u}{\tau}$. However, in all of the traces we have analysed, both k and r are much greater than 1.

The pathmap algorithm can easily be made more scalable by parallely computing the service graph of each client nodes (i.e., parallelizing the inner loop of *ServiceRoot*). The results reported in this thesis use a single central analyser.

2.2.8 Other Considerations

We have implicitly assumed that the clocks of all service nodes are time-synchronized. Pathmap can tolerate small clock skews (i.e., equal to few times of the time quanta τ) when determining service paths, but will exhibit some inaccuracy (equal to the amount of skew) when computing service delays. Fortunately, most of today's machines are synchronized using NTP, which has an RMS errors of less than 0.1 ms on LANs and of less than 5 ms on Internet (except during rare disruptions) [64]. If the skew is large, cross-correlation results will not be accurate. We can, however, estimate time skew between two service nodes (say x and y) by cross-correlating the time series $T^{x}_{x \rightarrow y}$ and $T^{y}_{x \rightarrow y}$ streamed from x and y respectively. The resultant cross-correlation series will have a spike at position 'd', where d is equal to the sum of the time by which x lags behind y and the network delay. The latter can be computed easily by one of the various passive network measurement techniques [50].

As discussed earlier, a single physical client machine may be modelled as multiple logical client nodes ($V_c s$), one each for the service class it generates. The network packets from these $V_c s$ to the front-end service nodes may have the same (source, destination) IP pair because of which it becomes difficult to distinguish one V_c from the other. An accurate request classification at the front-end server would require domain-specific knowledge, and toward this end, we are investigating generic kernel-level techniques (kernel TCP virtual server or ktcpvs [99], for example) that can parse request content to identify their service classes before forwarding them to the front-end server. No restart or modification of application components will be necessary.

The statistical approach of our service path algorithm detects the most probable paths and not the rare paths. The later requires more domain- specific knowledge and instrumentations and has been studied . Our result is immune to minor packet losses and retransmissions.

2.3 Experimental Evaluation

The E2EProf toolkit has been implemented in C and tested extensively on Linux-based platform for both artificial traces and actual enterprise applications. We will present results

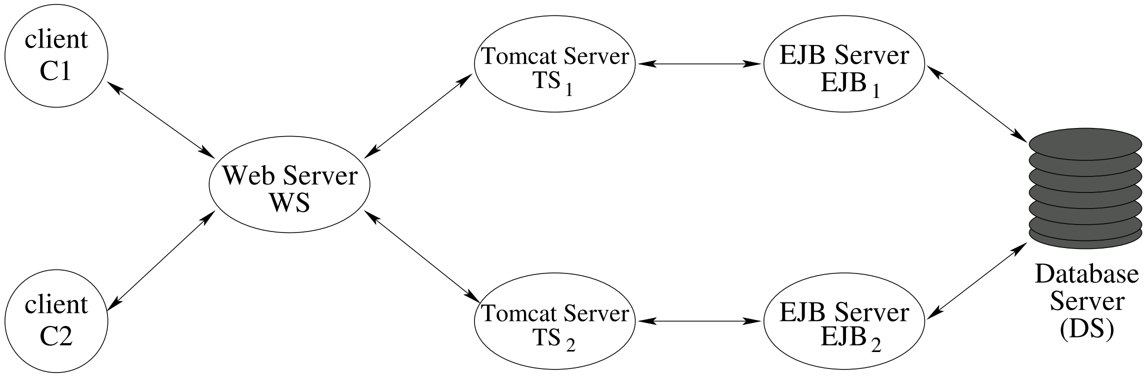


Figure 8: Multi-tier RUBiS application setup

from just two enterprise-scale multi-tier applications. The first is an open source multi-tier online auction benchmark, called *RUBiS*, from Rice University [26], and the second is the *Revenue Pipeline* application used by Delta Air Lines. We evaluate the overhead and accuracy of E2EProf and demonstrate how it can be used for online performance debugging in these applications.

2.3.1 Multi-tier Application: RUBiS

RUBiS implements the core functionalities of an auction site like selling, browsing, and bidding. RUBiS is available in three different flavors: PHP, Java HTTP Servlets and Enterprise Java Beans (EJB). We use the EJB's stateless session beans implementation with the following configuration:

- Front end: Apache 2.0.40 web server;
- Servlet Container: Jakarta Tomcat 5.5.9;
- JOnAS EJB server 4.4.3 using the *Jeremie* communication layer; and
- Database server: MySQL 4.1.14.

All servers are hosted on dual Intel Xeon 2.8 GHz, 512KB cache, 512MB RAM and connected via 1Gbit ethernet. Each of the machines runs RedHat Linux 9.0 (kernel version 2.4.20). 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 does not spend too much time forking threads at the start of each experiment;

- 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.

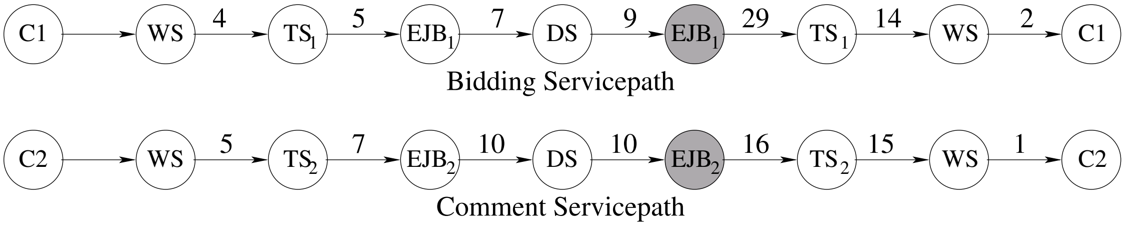
Figure 8 shows the RUBiS configuration used in all experiments. The *Tracer* kernel module runs on all six server nodes and streams time series data to a remote analyzer (not shown in the figure). The two client nodes run *httperf* [67] to generate requests belonging to two service classes (i.e., *bidding* and *comment*). The httperf workload generator in the client nodes emulates 30 clients by initiating 30 client sessions each. Web service requests generated by these client sessions have a *Poisson* arrival distribution. We experiment with two different path configurations:

- *Affinity-based*: the web server forwards all bidding requests to Tomcat server 1 (TS_1) and all comment requests to Tomcat server 2 (TS_2). The path of the bid request becomes $C_1 \rightarrow WS \rightarrow TS_1 \rightarrow EJB_1 \rightarrow DS$. Similarly, the path of the comment request is $C_2 \rightarrow WS \rightarrow TS_2 \rightarrow EJB_2 \rightarrow DS$.
- *Round-Robin*: the web server dispatches requests to the two tomcat servers in a round-robin fashion. Here, the bid requests take two different paths: $C_1 \rightarrow WS \rightarrow TS_1 \rightarrow EJB_1 \rightarrow DS$ and $C_1 \rightarrow WS \rightarrow TS_2 \rightarrow EJB_2 \rightarrow DS$. Similarly, comment requests has two paths: $C_2 \rightarrow WS \rightarrow TS_2 \rightarrow EJB_2 \rightarrow DS$ and $C_2 \rightarrow WS \rightarrow TS_1 \rightarrow EJB_1 \rightarrow DS$.

For RUBiS experiments, the pathmap algorithm parameters are configured as follows: Sliding Window (W) = 3 minutes, refresh interval (ΔW) = 1 minute, time quanta (τ) = 1ms and sampling window size (ω) = 50ms. The upper bound on transaction delay (T_u) is set to 1 minute. These values are chosen based on the guidelines discussed in section 2.2.4.

2.3.1.1 Service Path Detection

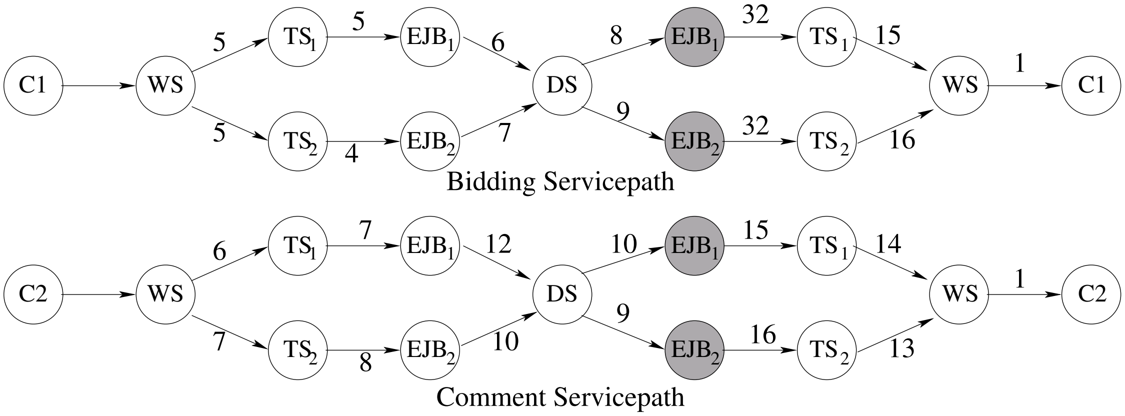
The goal is to demonstrate that E2EProf can detect these paths and their end-to-end delays, automatically, from the network packet timestamps collected. Figure 9 shows the service



Bidding Servicepath

Comment Servicepath

Figure 9: Service Graph for affinity-based server selection. (All delays in milliseconds)



Bidding Servicepath

Comment Servicepath

Figure 10: Service Graph for round-robin server selection (All delays in milliseconds)

graph for affinity-based server selection. Here, E2EProf correctly discover the paths of the two type of client requests. The vertices indicate the different servers, which are hosted on different physical machines. The label on the edge indicates the sum of the computation delay at the source node and of the communication delay from source to destination node. The paths of two types of requests are structurally similar, except for the difference in the service nodes they traverse and the delays incurred. The major sources of delay are automatically detected by E2EProf and marked in grey (i.e., the EJB servers in the figure). Note the duplicate vertex label in the service path. This is due to the return path taken by the response. For clarity, we avoid using cycles in the figure.

Figure 10 shows the service graph for round-robin server selection approach. The two paths taken by each type of requests are shown, and the major source of delay are marked in grey.

In order to verify the correctness of our results, we add code to RUBiS' servlets and EJB components to keep track of transaction latency at different servers, by piggybagging performance delay information in requests and responses. The resulting performance data

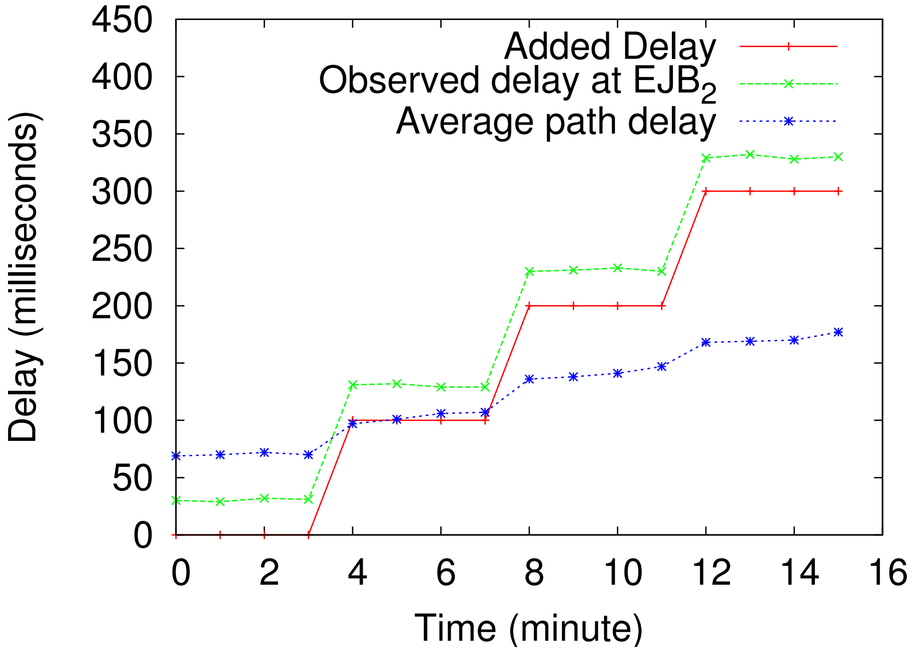


Figure 11: Performance change detection

coupled with the access logs from the web server and the response time observed at the clients are compared against the service path results generated by E2EProf. The difference of the processing delays computed at each server is within 10%. The latency observed at the client is about 16% more than that obtained from E2EProf.

The graph also shows two paths ($\cdots \rightarrow EJB_1 \rightarrow DS \rightarrow EJB_2 \rightarrow \cdots$ and $\cdots \rightarrow EJB_2 \rightarrow DS \rightarrow EJB_1 \rightarrow \cdots$) that don't exist in reality. They occur because the pathmap algorithm just establishes the fact whether request from $C1$ passed through the edge $DS \rightarrow EJB_1$ and $DS \rightarrow EJB_2$. It doesn't check whether the traffic in $DS \rightarrow EJB_1$ came via $EJB_1 \rightarrow DS$ or via $EJB_2 \rightarrow DS$. This ambiguity can be easily resolved by additional cross-correlation test at the vertices where the request path multiplex or demultiplex.

2.3.1.2 Change Detection

One of the goals of online service path analysis is to detect changes in path performance. We are interested not only in cumulative end-to-end delays, but also in fluctuations in *per-edge* performance. This is useful for isolating bottlenecks, re-routing request traffic, debug anomalies, etc. In order to demonstrate this capability of E2EProf, we vary the performance of one of the EJB servers (EJB_2) in the round-robin server selection setup, by artificially introducing some amount of delay in the bid request processing and increasing it after every 3 minutes. The length of the sliding window (W) is set to 1 minute. The other parameters of the pathmap algorithm are the same as in the previous experiments. Figure 11 shows the actual delay introduced and the bid request processing delay at EJB_2 captured by E2EProf. The algorithm correctly tracks the change in performance. The difference between the observed and added delay is due to the fact that the former includes the actual time spent by EJB_2 in processing the requests in addition to the artificial delay introduced in the experiment. The delay patterns of other edges remain unchanged. The figure also shows the average processing delay observed at the front-end web server. Since more than half of the requests take the low latency path (via EJB_1), the average delay does not change by the same amount. In cases like these, E2EProf can help diagnose bottlenecks faster, because it can separately track the performance of each service node.

2.3.2 Automated Path Selection

The front-end web server, among other things, has to perform request scheduling and dispatching, the purpose of which is to ensure load balancing and provide quality of service. Often, different workloads are associated with certain performance goals (e.g., minimum throughput or best response time) and may have certain SLAs associated with them. 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. Under normal circumstances, the round-robin server selection scheme works 'fairly' well. However, when the application servers experience performance problems, the simple round-robin scheme may not be able to meet SLA requirements, and the front-end web server has to intelligently choose the right back-end server for processing different classes of requests.

In order to improve upon round robin scheduling, we design a setup similar to the previous experiments, with two different classes of workload (bidding and comment), but introducing artificial delay experienced by the two EJB servers, which changes once per minute. These delays are randomly chosen, ranging from 0 to 100 milliseconds. The aim is

Table 1: Average latency with different path selection method

	Bidding	Comment
Round-Robin (No perturbation)	72 ms	64 ms
Round-Robin (with perturbation)	121 ms	109 ms
E2EProf (with perturbation)	97 ms	139 ms

to reduce the latency of the bidding requests. Furthermore, the server selection algorithm in the web server is modified to route bidding requests to the lower latency path and comment requests to the other based on path latency information obtained from E2EProf. Table 1 shows the average latency of bidding and comment requests measured during a 10 minutes period. After the perturbation is introduced, the average latencies of both types of requests increase with round-robin path selection. In comparison, the E2EProf-based scheduling method decreases the processing delay of bidding requests by directing them to the lower latency paths and penalizing comment requests.

The above is a straightforward example of automated performance management with E2EProf's path-based analysis. Clearly, the E2EProf-based path selection method performs better because it uses more information than the round-robin method, the latter being a black-box approach. We show these results simply to demonstrate E2EProf's utility for online and automated system management, in addition to its already proven use by system administrators to diagnose performance problems in complex enterprise applications. A concrete example of the latter is described in the next section.

2.3.3 Delta's Revenue Pipeline Application

The "*Revenue Pipeline System*" is a subsystem of Delta's OIS (Operational Information System) that keeps track of operational revenue from worldwide flight operations. It is composed of multiple black-box components (including legacy components) purchased from many different software vendors. The system is fed from multiple sources distributed worldwide, with events that represent ticket sales, passenger boarding, flight departures and arrivals, and others. Specifically, about 40K events per hour arrive in one of 25 queues in the front-end control system and are then forwarded to the back-end servers, as shown in

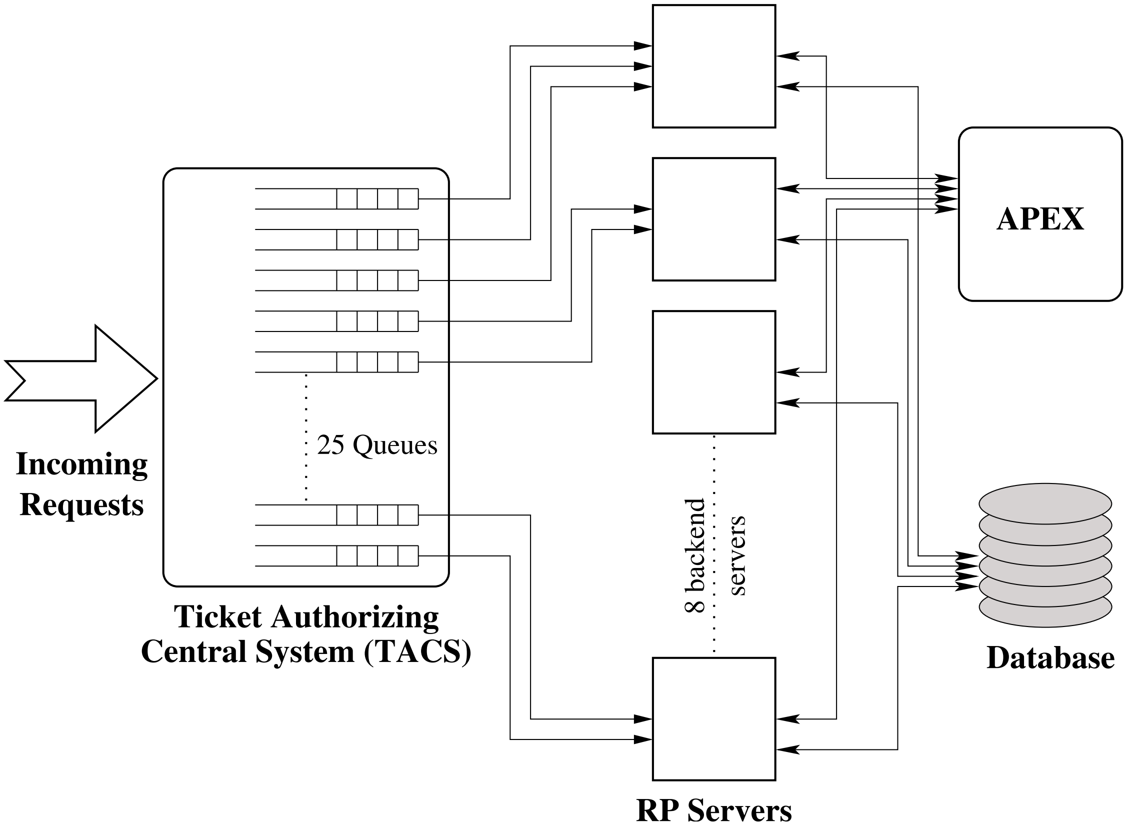


Figure 12: Delta Airlines' Revenue Pipeline Application

Figure 12. The complexity and significance of this subsystem can be estimated from the fact that Delta Airlines spends around US$ 40 million in a year to make sure that SLAs are met most of the time. Each event/request has strict SLAs. Missing a deadline can lead to financial costs in the form of productivity loss, lack of data for accurate financial planning and impediments associated with missing a mandated filing deadline. If an SLA is violated, system administrators have to analyze complex logs in order to isolate the faulty components. This process is quite time-consuming, in part because of complex dependencies across multiple black-box components.

E2EProf is used to analyse a week long trace collected from this subsystem. This trace consists of *access logs* from different servers and contains timestamps, server IDs, and request IDs for every application-level transactional event processed by the system (as opposed to the network-level packet events analysed in earlier experiments). The trace does not contain any personal or sensitive information like passenger names or credit card numbers, which are not meaningful for our analysis.

Several limitations of the existing pathmap algorithm are exposed by this use case. First, this subsystem's queuing delays can be large (much larger than the actual processing time). This changes the arrival pattern of the requests at different stages of request processing. Second, there can be wide variations in request traffic. For example, a batch process consisting of all of Delta Air Lines' paper tickets processed all over the world in the last 24 hours is submitted at 4 AM EST, due to which the queue length goes as high as 4000. These facts break the 'steady state' assumption made by the algorithm. Thus, although the pathmap algorithm is able to compute the service path correctly, the computed delays are far from accurate. In response, we have to carefully set the sliding window length (1 hour), the time quanta (1 second) and the sample window (50 seconds), thereby eliminating the error due to traffic variation. The analysis error due to the large queue length could not be eliminated.

Despite inaccurate delay computation, the service paths computed above are still useful in detecting causal dependencies across different components. For instance, E2EProf was able to successfully diagnose a slow database server connection that resulted in large

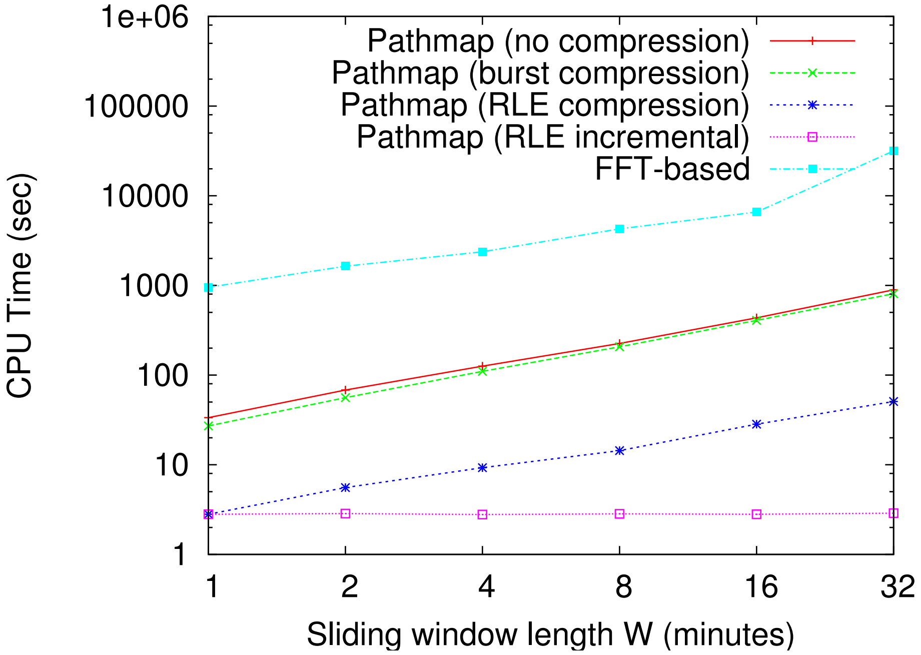


Figure 13: Execution time of service path analysis

response time for a moderate workload.

2.3.4 Micro-Benchmarks

Micro-benchmarks are used to examine the costs of E2EProf analysis for RUBiS traces. The results of overhead analysis for the Delta Air Lines traces were similar to those shown here. We evaluate the cost of E2EProf analysis with the different optimizations discussed in earlier sections and compare it with the FFT-based analysis. Figure 13 shows the time required to compute the service graphs shown in Figure 10 for different sliding window sizes (W). Other parameters of the pathmap algorithm are the same as in earlier experiments with RUBiS: τ = 1ms, ω = 50ms, upper bound on transactional delay(T_u) T_u= 1 minute. The plot labelled '*no compression*' just assumes an upper bound on transactional delay with no other optimizations. The '*burst compression*' plot only considers non-zero time series entries. '*RLE compression*' uses run-length encoded time series data.

From the results, it is clear that the RLE-based pathmap algorithm outperforms other methods by orders of magnitude. The cost of pathmap analysis increases linearly with W. For a sliding window of length 32 minutes, the RLE-based algorithm takes just 50 seconds.

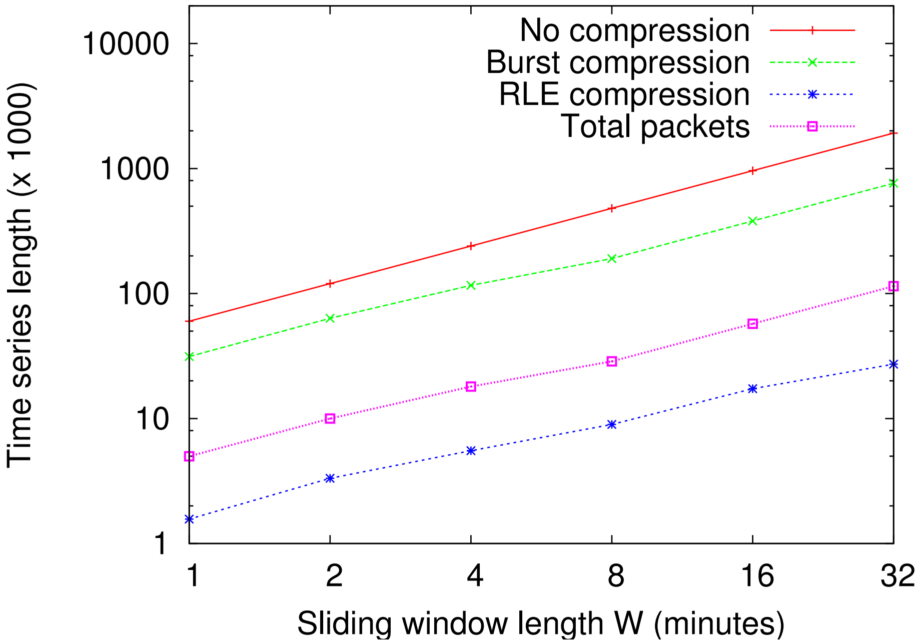


Figure 14: Time series compression

In reality, a 32 minute window may be too large for enterprise applications, as they need to react to changes within a few seconds to a few minutes. FFT-based analysis does not have linear cost and thus, takes an order of magnitude more time than pathmap to compute the same service graphs. Note that the cost of '*incremental*' pathmap analysis is almost constant for refresh interval (ΔW) set to 1 minute. This makes pathmap suitable for online analysis. The *burst compression* technique does not show much improvement over normal pathmap for RUBiS traces, but it decreases the length of time series (and therefore space overhead) significantly, as shown next.

Trace size: Figure 14 shows the compression achieved by different pathmap's optimizations for the time-series data of the connection between one of the tomcat servers and the web server. The plot labelled '*total packets*' shows the number of packets captured from which these time-series was computed. The time series length increases linearly with window size W, and the plot labelled '*no compression*' is the upper bound ($\frac{W}{\tau}$) on the time series length for a given W and τ. Once again, RLE compression achieves the best results and decreases the length of time series by an order of magnitude as compared to other optimizations. It is also much smaller than the raw timestamped data (indicated by the total number of packets). Although there are better techniques to compress packet traces, the advantage of using RLE compression is that it also reduces the time complexity of the pathmap algorithm.

2.4 Related Work

The large number of tools available for distributed system performance diagnosis may be categorized based on three broad features: online/offline, level of intrusiveness, and quality of analysis. They compromise one over the other to achieve their respective goals.

Single web server system performance has been studied extensively. EtE [41] and Certes [72] measure client-perceived response time at the server side. The former does offline analysis of the packets sent and received at the server side, while the latter does online analysis by observing the states of TCP connections.

Tracing tools for single systems like the Linux Trace Toolkit [98] and Dtrace [23] provide mechanisms for logging events by inserting instrumentation code. Compiler-level instrumentation is commonly used to understand program behaviors (e.g. gprof [44].) However, source code may not always be available, and the sizes and complexities of sources are disincentives for software engineers engaged in post-development instrumentation or evaluation. Even binary instrumentation requires some level of understanding of application details.

Path-level analysis of distributed systems tracks the causal relationship between different components and has recently been an area of active research. ETE [49] uses application-specific instrumentation to measure the latencies between component interactions and relates them to end-to-end response times to detect performance problems. Pinpoint [29] detects system components where requests fail, by tagging (and propagating) a globally unique request ID with each request. Magpie [19], on the other hand, requires no global ID, and it can capture not only the causal paths, but also monitor the resource consumption of each request. Industry standards like ARM [14] (Application Response Measurement) used by HP's Openview, IBM's Tivoli, and BEA's Weblogic require middleware-level instrumentation to measure end-to-end application performance. In contrast, E2EProf does not require any modification to applications and therefore, can also be used with legacy components. However, unlike Magpie, it does not measure general resource usage.

The work by Aguilera *et al.* [9] is most closely related to E2EProf. They propose two algorithms to determine causally dependent paths and the associated delays from the message-level traces in a distributed system. While their *nesting* algorithm assumes 'RPC-style' (call-returns) communication, their *convolution* algorithm is more general and does not assume a particular messaging protocol. Our pathmap algorithm is similar to the *convolution* algorithm, in that both uses time series analysis and can handle non-RPC-style messages. While the convolution algorithm is primarily intended for offline analysis, pathmap uses compact trace representations and a series of optimizations, which jointly, make it suitable for online performance diagnosis.

2.5 Summary

The complexity of distributed systems have been increasing rapidly. To address this complexity, our research has developed a toolkit for online, end-to-end performance diagnosis of distributed systems, called E2EProf. The toolkit uses a modified form of time-series analysis (commonly used in Digital Signal Processing or DSP), to detect the paths taken by requests and delays incurred due to different path components. Since the toolkit does not require applications to be modified, it can also handle legacy components. Experimental evaluations show that E2EProf can detect performance bottlenecks in realistic enterprise applications, while at the same time, reducing the analysis time by an order of magnitude compared to similar techniques presented in the literature.

Our near term future work will explore other areas and applications to which the techniques presented in this thesis can be applied. These include network overlays and publish-subscribe systems [56, 82, 83]. Further, we have recently been able to start a collaboration with another group at Delta Air Lines that manages the Delta.com infrastructure, which is much more complex than the revenue pipeline system. Analyzing these new traces will provide us with new insights into the challenges posed by complex enterprise applications. We are also building visualization interfaces that would highlight interesting performance behaviors of service paths. In the long term, we plan to deploy E2EProf as a basic service, ‘pluggable’ into any distributed system. When applications or services subscribe to its interfaces, they henceforth, will receive real-time information about their service paths and systems ‘health’ in general.

CHAPTER III

SYSPROF: ONLINE FINE-GRAIN SYSTEM MONITORING

3.1 Introduction

The E2EProf toolkit described in the previous chapter discovers and analyze the end-to-end service paths and the dependencies across different services. This is useful for debugging performance problems in scenarios where its not possible to modify system components. Runtime management of enterprise applications, however, often requires detailed performance analysis, going beyond measurements of simple metrics like average CPU load, network bandwidth, number of tasks completed, etc. [17]. One approach is to integrate generic methods for analyzing program performance into middleware, used in systems like Photon [89], Pinpoint [30], and many others [92]. The idea is to automatically observe a program's usage of middleware functions, including the middleware-mediated interactions between different, distributed application components. Applications need not be modified, and access to source code is not necessary. However, since actual resource usage is controlled by the operating system, it is not possible to accurately account for the performance effects of certain application- or middleware-level behaviors. The basic causes of these problems are system-level asynchrony, i.e., the OS kernel's internal use of concurrency to satisfy multiple application requests, and system-level independence, i.e., the fact that OS kernels independently manage and allocate system resources for the multiple application-level processes being run. From the middleware level, therefore, it is difficult to attribute the usage of certain system resources to specific user-domain actions.

Prior research work has already recognized the importance of making operating systems more flexible and accountable for their resource usage [18, 65]. The goal of our research is to provide to applications accurate and timely information about their current resource usage. This chapter describes the *SysProf* system-level toolkit, which provides a flexible framework for refining the E2EProf's service path analysis and measuring the resource consumption behavior of various activities within a service path. An *activity* may be a *system call* made by some user-level application, or it may be a specific request-response interaction between a client and a web service. An activity may also be some class of application-level actions, such as the composite behavior of requests residing in a high priority request queue in an application server. In all such cases, SysProf provides support for carrying out enterprise-wide measurements – from application to system levels and across multiple machines – of the resources used by activities. SysProf's interface is such that activity monitoring may be customized, at runtime, to current needs. Furthermore, with the monitoring of runtime activities may be associated the analyses needed to aggregate, filter, or correlate monitoring data, as per current diagnostic needs. Analyses are carried out by pre-built kernel-level functions that can be dynamically activated or de-activated, and/or they can use custom functions specified by the application or system administrator. Furthermore, after local, in-kernel analysis, monitoring data may then be aggregated and sent to remote analyzers (or to any remote data consumer) through kernel-level publish-subscribe channels. These channels potentially connect all machines participating in the activities being carried out. In essence, therefore, SysProf uses a system-level overlay to capture, analyze, and correlate monitoring data. The overlay's actions may be dynamically customized to meet the granularity and real-time needs of the processes that require monitoring information.

SysProf does not require changes to user-level code, including changes that would re-compile it (e.g., with a debug switch). By using system-level mechanisms for monitoring user-level applications, SysProf can run without user involvement and without source code knowledge. Another advantage of SysProf is its ability to collect richer and more accurate information than is possible at user level. This includes tracking in detail the actions of specific dynamically selected applications, application components, and properties of their behaviors.

SysProf extends our earlier work on kernel-level monitoring, termed DProc [6, 7] and makes the following new contributions:

- SysProf provides a flexible framework for monitoring at the granularity of individual *activities*, such as the system calls issued by a specific client or a client's interactions with a certain remote application service.

- SysProf's analysis actions associated with the runtime capture of monitoring data are configurable dynamically, thereby enabling tradeoffs between the granularity, overheads, and delays of runtime diagnoses.
- High performance and low perturbation for low granularity monitoring are due to SysProf's use of dynamic code generation, binary encodings for monitoring data, low overhead kernel-level publish-subscribe messaging, and efficient event hashing.
- The utility of SysProf is demonstrated in two application contexts:(i) in a shared NFS service where SysProf can dynamically detect bottlenecks in proxies vs. servers, and (ii) in a multi-tier auctioning web service called RUBiS [26], where SysProf-based runtime monitoring and diagnosis are used to improve the scheduling of client requests.

Micro-benchmarks and performance evaluations of SysProf validate the importance of low granularity and highly accurate monitoring. The overhead of SysProf is within acceptable limits that makes it possible to be applied to many online algorithms. In our evaluation, application performance of an online E-Commerce website decreased by less than 2% because of SysProf. But the throughput gain ($> 14\%$) that was achieved with SysProf far outweighed the cost. SysProf was also able to determine the bottlenecks in a virtual storage service by correctly identifying the sources of latencies in the system.

The remainder of this chapter is organized as follows. The next section describes the design principles of *SysProf*. Section 3.3 outlines the architecture and implementation of its various components. Section 3.4 first explains the innate overheads of SysProf. Next, the importance of low granularity, per request monitoring is demonstrated with a use of SysProf that dynamically detects bottlenecks in a shared NFS service. Finally, the ability of SysProf to provide data with low delay enables its use for implementing resource-aware scheduling in a multi-tier web service. Section 3.5 describes related work, and summary appear in Section 3.6.

3.2 SysProf: Design and Architecture

The **SysProf** toolkit keeps track of the different *activities* in a distributed system and resources consumed by them. An *activity* may involve just one machine, like a system call that reads file data from a local disk, or it may span multiple machines, like a "HTTP" request in a multi-tier web service. In either case, an activity is a SysProf-defined entity that is not constrained to match a single application-, middleware-, or system-level abstraction. This thesis focuses on activities that involve network interaction between multiple machines. Our ongoing work is using the activity notion to better understand end-to-end application properties in the light of concurrent OS behavior on single machines.

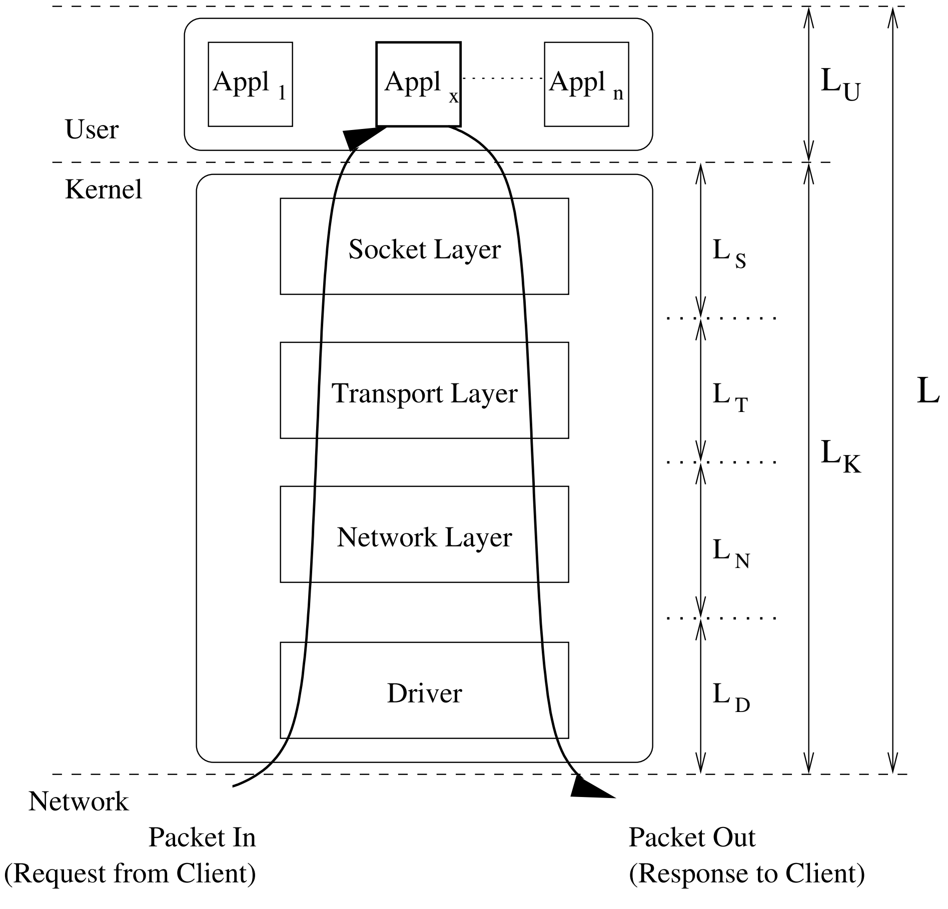


Figure 15: An Activity example: Different L's show the time spent (latency) at each of the marked steps

Figure 15 presents a sample activity. A request packet from a client arrives in a system and after being processed by different network protocol layers, it is delivered to the user-level server , Appl$_x$. The server performs some computation on the request and calculates a response, which is then sent back to the client, after again traversing the network stack. At each processing step, some resources (e.g. CPU cycles, memory, etc.) are used, and the request may be queued a number of times before a response is finally sent out. In order to debug the performance of this application and detect potential bottlenecks, the developer or the system administrator may need to know the time spent and resources consumed at each of these steps. In addition, the developer may need to understand queuing and concurrency behaviors. SysProf can provide details about the time spent in different steps of the network protocol processing, (the different L$_i$ values in Figure 15); time spent by application at the user-level and at the kernel-level while the request is being processed, time spent by the application waiting for I/O during request processing, etc. In addition, SysProf can also maintain mean, variance, and other statistical metrics as required. Further, such SysProf monitoring requires no modification to the Appl$_x$ while providing detailed insights into the execution of the client's request. (1) Monitoring information can be used to identify bottleneck resources, by identifying where most of the time is spent (i.e., at the kernel-level or at the user-level). (2) It can identify the reason a request spends some unusual amount of time in the kernel buffer, perhaps because there are too many outstanding requests Appl$_x$ must process or perhaps because of some bug in the Appl$_x$ itself. (3) It can identify what Appl$_x$ was doing when the request was waiting in the kernel buffer? Was it executing or was it blocked for some reasons (e.g. I/O)? Answers to such questions are important steps toward identifying performance problems in networked IT infrastructures.

The depiction of the communication activity shown in Figure 15 is oversimplified. Actual web service requests, for example, may be processed locally by the server by fetching data from local disks, or they may query database servers on remote machines before responses are generated. Such requests may be processed asynchronously by processes different from the ones who originally received them (e.g., proxies), and control transfers may be accomplished by shared memory, message queues, or with other IPC mechanisms. Other issues like concurrency (to handle requests from different clients) and interleaving (handling different requests from same client) further complicate request analysis. We address these issues and discuss the assumptions and limitations of SysProf in the next few sections.

3.3 SysProf: Software Architecture

The SysProf architecture has been designed with the following goals in mind:

- Zoom into the service path discovered using E2EProf for fine-grained resource analysis;
- No changes to user-level code;
- Customizability: the ability to dynamically tune monitoring and analysis, including which resources to monitor, monitoring frequencies, the metrics to be generated (variance, mean), etc.;
- Extensibility: the capability to dynamically upgrade or add new monitoring and analysis functionality to react to changes in requirements, infrastructure, or for other reasons;
- Low overhead: small perturbation for each measured activity or service;
- Heterogeneity: the ability to run on and interoperate across different hardware and software platforms; and
- Standard API: provision of a common API through which local and remote services can access all information about monitored activities.

Figure 16 depicts an overview of the SysProf architecture. It has five main components, which are described in detail in the next few sections.

- *Kprof*: Kernel instrumentation and logging;
- *LPA*: Local Performance Analyzer;
- *GPA*: Global Performance Analyzer;
- SysProf Dissemination Daemon; and

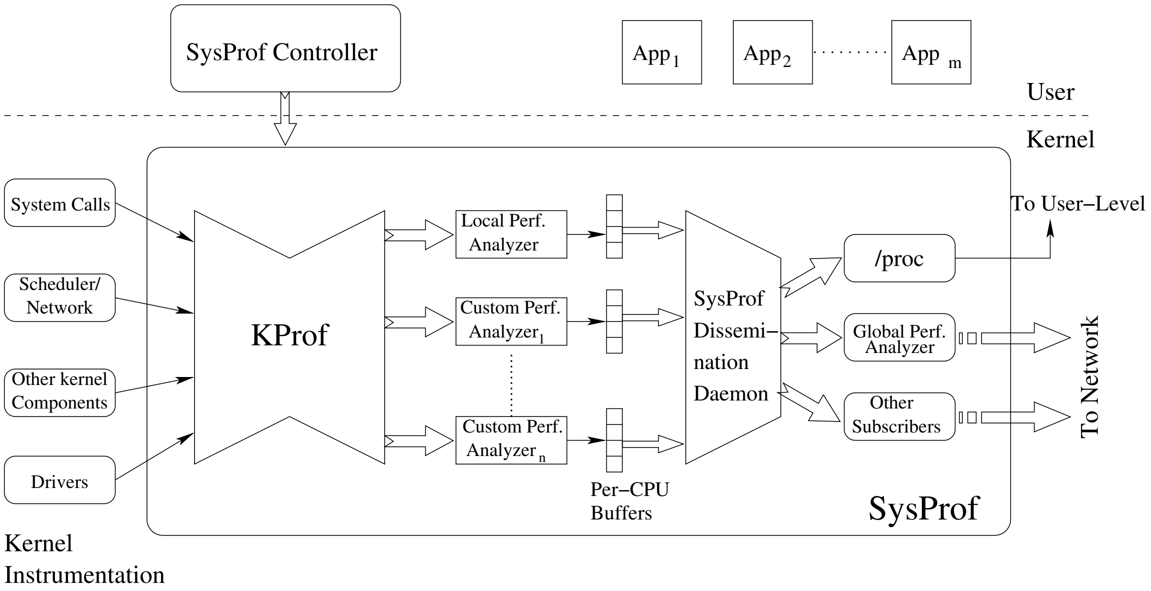


Figure 16: SysProf Software Architecture

- SysProf Controller.

These components are described in more detail next.

3.3.1 Kprof

Kprof is the SysProf monitoring interface. It operates at kernel level and provides a generic API for the collection of various events from different kernel components. To track activities, a set of key points in the kernel are instrumented statically (like in Linux Trace Toolkit [98]). Kprof receives information from these points as efficient binary events. Events are delivered by invoking a function provided by Kprof's API. These events can be grouped into four major types: Scheduling events (context switches, process creation/deletion, etc.), System Call events, Network events, and File System events (open, close, read, write, etc).

Figure 17 shows some of Kprof's APIs. *Struct _event* is the common structure of the event generated by the Kprof instrumentation. A private variable points to event-specific data. The figure also shows the type of the event (*struct _pkt_out_event*) that is generated from the network driver level when the packet is sent out on the network. Among other attributes, it notes the time when the data contained in this packet was copied to the kernel buffer from the user-level. This timestamp is the basis for computing the amount of time spent by this response packet in the kernel.

```
#define KPROF_EVENT(event) \
      do{                           \
           if (event.ID & flag)     \
                   kprof(event);    \
      }while(0)

struct _event{
      unsigned int ID;
      unsigned int sub_event_ID;
      unsigned int timestamp;
      unsigned int cpuID;
      unsigned int PID;
      void *private;
}

struct _pkt_out_event{
      unsigned int curr_timestamp; /* Time when this
                                      event was generated */
      unsigned int sock_stamp; /* Time when the application
                                  put this buffer in the kernel */
      unsigned int src_ip;
      unsigned short src_port;
      unsigned int dest_ip;
      unsigned short dest_port;
}
```

Figure 17: Kprof Monitoring API

Events can be selectively switched *on* and *off* depending on the requirement set by the SysProf controller or the *local performance analyzer* (LPA). Events can also be pruned on the basis of process IDs, group IDs, or other such predicates. Each LPA specifies the set of events in which it is interested by registering a callback function with Kprof. These callbacks are invoked by Kprof when their events are generated. When none of the analyzer(s) subscribes to events, all of them are turned off, resulting in almost negligible perturbation for Kprof-instrumented operating system kernels. Kprof builds on our earlier dProc kernel-level monitor, and its functionality is similar to the static kernel instrumentation offered by LTT [98]. Further, using Kprof does not prevent us from using other available instrumentation techniques like Dprobes [66], Dtrace [23], Kerninst [86], etc. Our goal is not to innovate in kernel-level monitoring, but instead, to have sufficient facilities for extracting relevant monitoring information from OS kernel, without major kernel modifications and with acceptable perturbation(i.e., to avoid changing the behavior of the activity being measured [60]). The performance analyzer described next is more interesting.

3.3.2 Local Performance Analyzer (LPA)

The **Local Performance Analyzer** filters, aggregates, and correlates raw monitoring data, and then uses it to generate different performance metrics from the events generated by Kprof. There can be more than one LPA in SysProf, each potentially performing different analyses. During initialization, each LPA registers a callback with Kprof, and it specifies a list of events that need to be delivered to it. These callbacks are in the "fast path" of the kernel code and may also be invoked from interrupt contexts. Therefore, it is necessary that they never block and are computationally small. For lack of space, we next describe only one LPA in detail, the one that diagnoses a request-response interaction between a remote client and a user-level application server.

Messages and Interactions: The first diagnosis step is to identify a certain request-response pair. Because of interleaving and concurrency (as discussed in Section 3.2), it is non-trivial to extract such a pair without any application-specific knowledge. Recently, some offline black-box approaches have been proposed to infer causal path patterns [9], but online black-box techniques pose challenges like overhead and timeliness. In order to enable online analysis, SysProf defines the notions of *messages* and *interactions*. Let node$_A$ (identified by {node$_A$ IP, node$_A$ port} pair) and node$_B$ (identified by {node$_B$ IP, node$_B$ port} pair) be the nodes communicating with each other. A series of packets from node$_A$ to node$_B$ without any intervening packets in the opposite direction constitute one *message*. An *interaction* consists of a message pair in the opposite direction. Figure 18 shows a sample interaction and message pair. The intuition behind this approach is that requests and responses will be composed of multiple packets, where M$_1$ and M$_2$ in the figure correspond to one request/response pair[1].

LPA subscribes to multiple Kprof events to keep track of *interactions* and the different

---

[1] Multiple requests may interleave, in which case other techniques like time-series analysis or even data mining may be useful. However, they can be quite expensive for online analysis. Domain-specific (application or middleware) knowledge and/or ARM support [14] can be used with little overhead and we plan to study them in our future work.

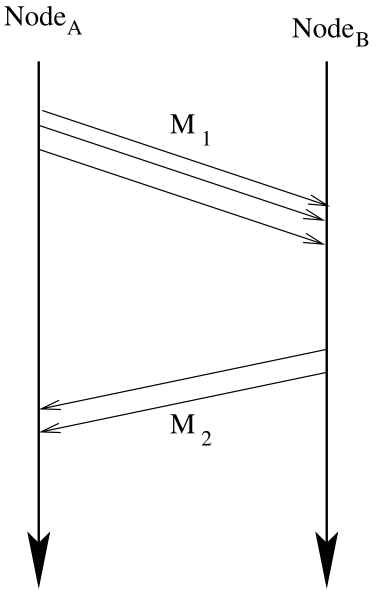


Figure 18: SysProf: Messages and Interaction

performance metrics associated with them. Specifically, LPA maintains a window containing the past several interactions and the metric values computed for them. Window size can be changed dynamically, and window contents are evicted to the dissemination daemon after some time. That is, each LPA maintains two per-CPU buffers to store captured data, and when one of them has been filled, the dissemination daemon is notified, and the LPA switches to the next buffer. Each such buffer switch requires interrupts to be disabled locally to avoid data corruption.

Information collected from these events includes the time at which some interaction started, the number of packets/bytes exchanged, the amount of time spent by the interaction in user and kernel modes, the interaction id, the name and the function of the user-level application server that receives packets from the interaction, and others. It is also possible to capture information about context switch details, the number of disk I/O operations performed by the application, and the length of time the application is blocked (e.g., for I/O) during an interaction. Such information capture can be configured and turned on and off dynamically, depending on current analysis requirements.

Custom Local Performance Analyzer (CPA): In addition to the statically defined LPAs, custom analyzers can be dynamically created and downloaded into the kernel. CPAs function just like normal LPAs, including registering of callbacks with Kprof and indicating the set of events they wish to receive. CPAs are specified in the form of E-Code [36] (a language subset of C), compiled through run-time code generation. CPAs provide great flexibility in terms of specifying application-specific analyses. A potential danger is that developers create CPAs that perform complex analyses and therefore, have large overheads. Fortunately, there are now well-known techniques [42] for executing such codes that guarantee restrictions in resource usage and in running time.

3.3.3 SysProf Dissemination Daemon

The **SysProf dissemination daemon** distributes the information generated by LPAs to the remote nodes that need it and also makes it available to the user-level through the standard "/proc" virtual filesystem interface (i.e., as with Dproc [6, 7]). On receiving a "*buffer full*" notification from a LPA, the daemon wakes up and copies the LPA's data into its own buffer. If the data is not picked up in a timely fashion, it may be overwritten. The size of the buffer, therefore, must be chosen carefully. Remote nodes subscribe to the information generated by LPAs, and it is the daemon's job to aggregate data collected from different LPA buffers in order to send it to interested parties. For high performance and low overheads in event acquisition and dissemination, the daemon uses dynamic data filters, PBIO-based binary encodings, and kernel-level publish-subscribe channels [6, 7].

3.3.4 Global Performance Analyzer (GPA)

The **Global Performance Analyzer** aggregates and correlates the data it receives from different SysProf daemons. Specifically, it correlates the source and destination IP addresses, port information, and NTP timestamps in the logs from different nodes After aggregating the resource usage of each individual interaction, GPA computes the overall performance of the associated request-response pair. Other nodes in the system can query the GPA to determine information about a particular interaction or about the system as a whole. The GPA periodically dumps its information onto local disk, which can be used later for purposes of auditing, workload prediction, and system modeling.

3.3.5 SysProf Controller

The **SysProf controller** regulates the granularity and the amounts of information monitored and analyzed by SysProf. It can instruct the LPAs to collect statistics for some client class rather than for individual interactions. It can change the sizes of internal LPA buffers. It provides a management interface for SysProf and makes it easy for the user to select its functionalities.

3.4 Experimental Evaluation

SysProf has been implemented in Linux (kernel version 2.4.19) as a set of loadable modules and a kernel patch that defines the instrumentation required to generate events, as discussed in Section 3.3.1. The current implementation only supports Intel x86 platforms, but the general technique is applicable to other architectures. The next few subsections describe the results of micro-benchmarks that assess the accuracy and overheads of SysProf. We then present our experiences with SysProf in detecting bottlenecks in a shared NFS application and in making scheduling decisions in an online auctioning web-site called RUBiS [26].

3.4.1 Microbenchmarks

As discussed in the previous sections, SysProf has a negligible effect on the performance of services it monitors. Because of its configurable interface, the overhead of SysProf can be varied ranging from less than 1% of the system resource to more than 10%. We measured the overhead in its default configuration (Section 3.3) by running it with *linpack* benchmark on a setup of two nodes (2.8GHz uni-processor, 512KB cache and 4GB RAM) connected to each other by a 1Gbps ethernet. *Linpack* measures the computation power of a machine in *MFLOPS*. There was no change in the *mflops* measured by linpack due to SysProf. One of the reasons is that SysProf generates more activities when there are network interactions, so linpack was probably not a very good benchmark. In another microbenchmark experiment, we employed a network bandwidth measurement tool called *Iperf* [2] to test the overhead of SysProf. Bandwidth was measured between two nodes, first with SysProf disabled and later

---

[2] Iperf Version 2.0.2: http://dast.nlanr.net/Projects/Iperf/

enabling it. The measured bandwidth in the later case (~810 Mbps) was almost 13% less than that of the former (~930 Mbps). This reduction in bandwidth was due to overhead incurred by examining packets at such high speed and not due to SysProf network usage. In a 100Mbps LAN, this overhead came down to 3%.

3.4.2 Shared NFS Proxy

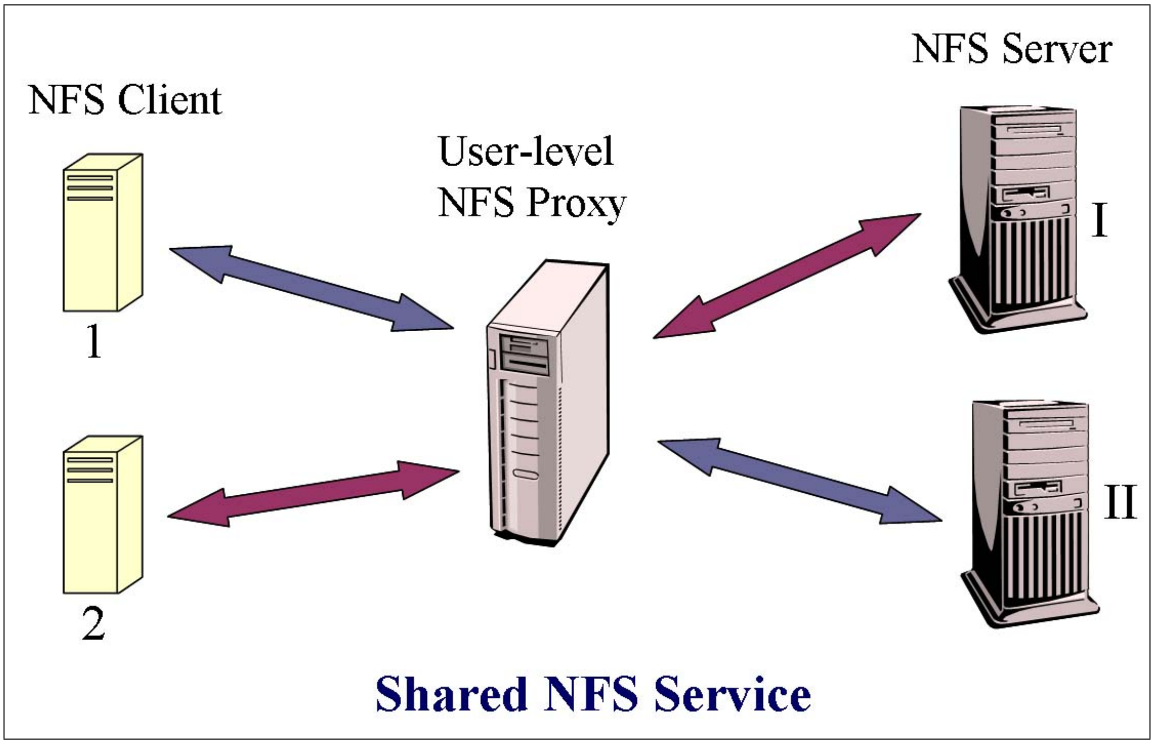


Figure 19: Virtual Storage Service

Virtual storage architecture has been quite popular in the enterprise storage domain to provide fast, efficient and fault tolerant service to the consumers [76, 5, 11]. The virtualized architecture simplifies many management problems in storage system. The user has a single front-end view of a large system at the back end [57]. New disks may be added or removed dynamically without service disruptions. Recent research has proposed various techniques to deliver enhanced quality of service including performance isolation and service differentiation [55, 59]. Figure 19 shows a very simplified version of a virtual storage service. The back-end storage servers are hidden from the client's view by a user-level proxy that interposes every request from the client to the server. The real scenario is of course very complex and may consist of multiple level of hierarchy, each providing some kind of service and with multiple front-end proxies. A typical problem in these environments is to detect failures and performance bottlenecks. One of the ways to detect this is by tracking the execution of the requests through different components and measuring the latencies and resources consumed. However, this is a difficult problem because there may be a large number of nodes involved through which the requests pass and get processed.

In this section, we illustrate how SysProf can be used to detect performance bottleneck in a virtual storage service like the one shown in Figure 19. Each of the machines has dual Intel Xeon 2.8 KHz processor with 512KB cache and 4GB memory and connected via 1Gbps switched ethernet. The proxy and the servers ran RedHat 9.0 with 2.4.19 kernel patched with SysProf. The client machines ran Red Hat 9 stock kernel. We ran a filesystem benchmark called *Iozone*[3] as client's workload. *Iozone* benchmark can test the performance of the number of I/O operations. We configured *Iozone* to generate write/re-write tests and varied the number of threads it forks to see the effect on resource usage. The number of threads created in each runs were same for both the clients.

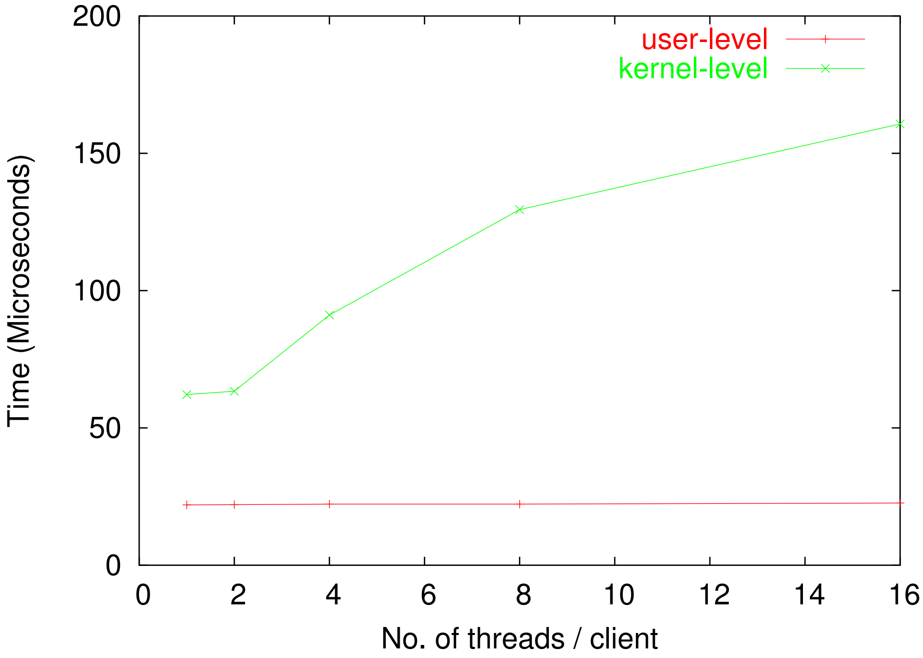


Figure 20: Avg time spent by client-proxy interactions at the proxy

---

[3] http://www.iozone.org/

Figure 20 shows the average amount of time an *interaction* (as described in Section 3.3.2) between client and proxy spend at the proxy node, both at the user-level and at the kernel-level. The amount of time a request spent at the user-level is almost constant for different number of client threads but the kernel time goes up because of increase in the request traffic. This is because the proxy does very little processing of requests and its job is to just forward the request to the back-end NFS servers. Therefore, it spends a constant amount of time on every request it processes. But as the traffic is increased, kernel buffers get filled up and the requests get queued at the kernel-level waiting for their turn to get processed by the user-level proxy. It should be noted that the time shown in the above figure is the time spent within the proxy node only and doesn't consist of time spent in the network or other nodes.

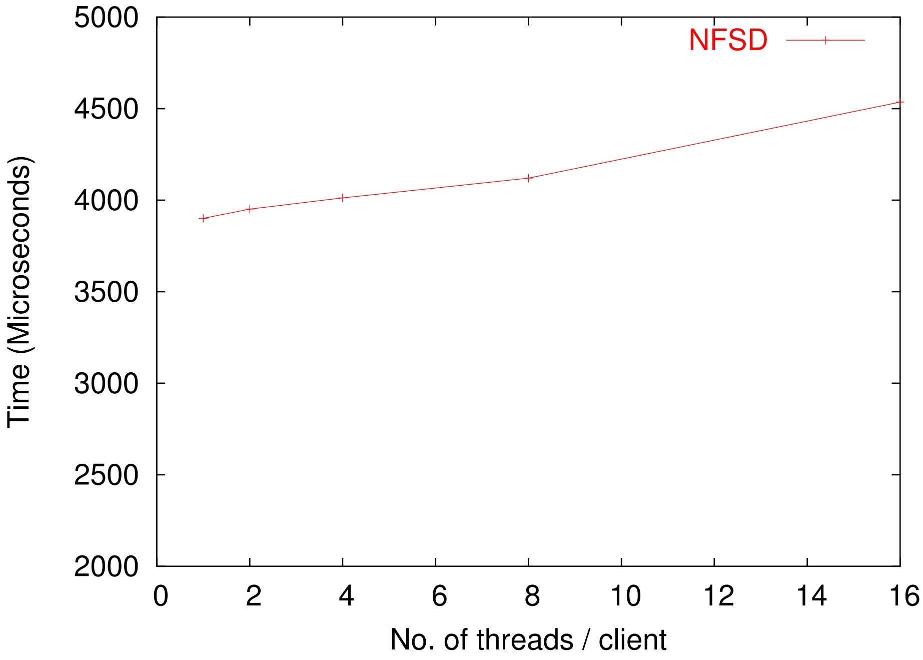


Figure 21: Avg time spent by the interactions at back-end server

We did a similar measurement with SysProf at one of the back-end NFS servers. Since the NFS server ran as kernel daemon, no time was spent by the request at the user level. Figure 21 shows the average time spent by different *interactions* in the kernel of the back-end server. This time is more than an order magnitude than the time spent in the proxy. This shows that the the back-end server is the major contributor to the delay seen in processing the client request. The network round-trip delay is insignificant ($< .3$ms) as compared to the time spent in the back-end.

From the above study, we showed how SysProf can be used to identify the bottleneck resources. It not only tells the delay incurred in request processing on a particular node but also gives fine details like whether the amount of time was spent in user-level or kernel-level, the number of outstanding interactions and so on.

3.4.3 Multi-tier Web Service

In this section, we study a dynamic window-constrained scheduling algorithm for a multi-tier web application called RUBiS, and show how it can provide better QoS using the information provided by SysProf.

Application Description. Modern enterprise applications are usually multi-tier architectures that are composed of number of tiers each providing certain services. A client's request is processed by some or all of these tiers before a response is generated. A typical e-commerce site consists of a web server at the front-end, a number of application servers in the middle tier and database servers at the back end [24]. In our experiment, we employ an open source online auction benchmark called RUBiS [26]. RUBiS implements core functionalities of an auction site like selling, browsing and bidding. RUBiS is available in three different flavors: PHP, Java HTTP Servlets and EJB. We use the Servlets version with the following configuration:

- Front end: Apache 2.0.40 web server;
- Servlet Container: Jakarta Tomcat 5.5.9; and
- Database server: MySQL 4.1.14.

All the servers were hosted on IBM Blades with dual Intel Xeon 2.8 GHz, 512KB cache, 4GB RAM and connected via 1Gbit ethernet. Each of the machines ran RedHat Linux 9.0 (kernel version 2.4.19) with SysProf extensions. The servers ran in their default configuration except the following settings:

- *MaxSpareServers* of the Apache web server was increased to 50 so that the server doesn't spend too much time in forking the threads at the start of each experiments and effect 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.

Figure 22 shows the setup that we used in the experiments in this section.

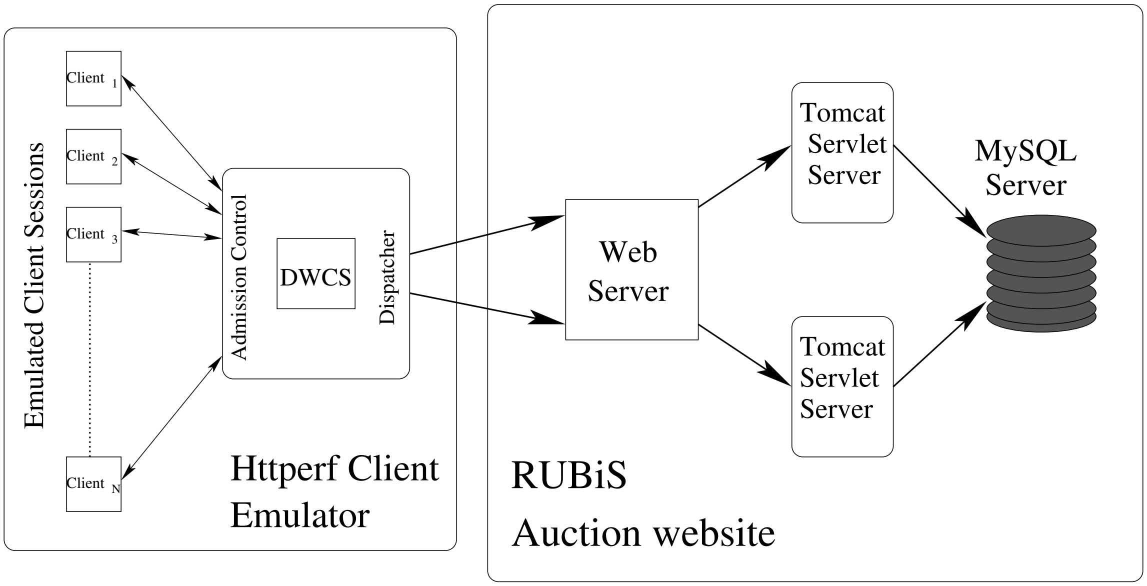


Figure 22: Three-tier e-commerce website benchmark: RUBiS (Rice University Bidding System)

The front end web server, among other things, has to do request scheduling and dispatching, the purpose of which is to ensure load balancing and provide quality of service. Lots of state-blind (i.e. black-box) and state-aware request scheduling and routing algorithms have been proposed in the literature [15, 12, 22, 54]. The advantage of using black-box approach is that they require no modification to the back-end and usually incur very little overhead as compared to state-aware algorithm. However, they usually operate under very strong assumptions about the resources available at the back-end and can make very poor decisions. In this evaluation study, we apply a black-box scheduling algorithm called *DWCS* to

RUBiS and demonstrate that a *resource-aware DWCS* can provide better QoS guarantees as compared to the ordinary *DWCS*.

Dynamic Window-Constraint Scheduler (or DWCS) [94] is a real-time scheduler based on three attributes: a period T, a window-constraint or loss rate x/y, and a run time C, where DWCS guarantees an activity C time units of service within a period T. However, this guarantee is relaxed by the loss rate, which indicates that x service invocations in y consecutive periods (i.e., $y * T$ time units) can be missed. If a packet is not scheduled within a period T, it is said to have missed its deadline. If the number of missed deadlines exceeds x in a window of y, the stream is said to have suffered a violation. The adjustable parameters of a DWCS stream are the period and the loss-rate. Although DWCS has traditionally been used in streaming multimedia applications that can often tolerate infrequent losses or misses of data generation or transmission and in linux process scheduling [93], it is equally applicable in enterprise domain where different workloads need to be multiplexed in a shared utility infrastructure (like a multi-tier web service). These workloads are often associated with some performance goals (like the minimum throughput or the maximum response time) and may have certain real-time requirements which are usually expressed in the form of Service Level Agreements (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 deadlines.

Table 2: Precedence among Pairs of Requests in Different Classes

Pairwise Request Ordering
Earliest Deadline First (EDF)
Equal deadlines, order lowest window-constraint first
Equal deadlines and zero window-constraints, order highest window-denominator first
Equal deadlines and equal non-zero window-constraints, order lowest window-numerator first
All other cases: first-come-first-serve

DWCS can generate schedules which ensure that these deadlines are met as well as achieve service differentiation and performance isolation between these workloads. In this new domain, the notion of *packets* is changed to *requests* and that of *packet streams* to *request classes*[4]. Precedence is given to requests in classes according to the rules shown in Table 2. Once a request is selected to be scheduled, DWCS forwards it to a back-end application server with minimum number of outstanding response. The maximum number of responses that can be outstanding at any application server is fixed and is determined statically on the basis of server's capacity. DWCS is work-conserving and requires no special support from the backend.

We apply DWCS to schedule two different request classes in RUBiS setup of Figure 22 with SysProf disabled. These requests were generated using *httperf* [67] on a separate client machine with the same configuration as other server machines. 60 client sessions were created and half of them generated high priority ($x/y = 1/30$) *bidding* requests and the other half generated low priority ($x/y = 1/10$) *comment* requests. The *bidding* request is cpu intensive and consumes lot of cpu at the servlet server which processes it. The *comment* request on the other hand generates significant network traffic. Each request class has a *Poisson* arrival distribution with mean rate equal to 150 requests/sec. The scheduler ran on the same node as the client and the request dispatching was facilitated by prefixing the request's URL path with the appropriate servlet server's name. Apache server was configured to multiplex the requests to the different backend server depending on these prefixes.[5]

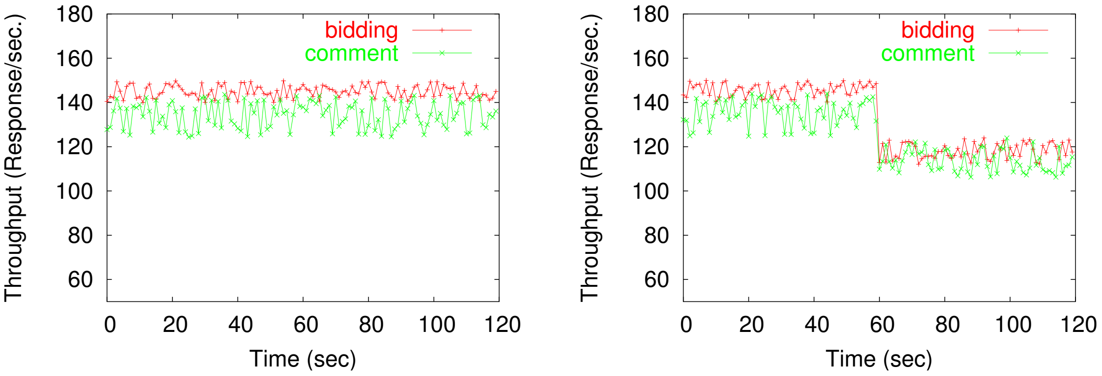
Figure 23(a) shows the result of applying DWCS to the two request classes. The average throughput achieved for *bidding* requests and *comment* requests were 145 and 134 responses/sec. respectively. This throughput excludes responses that missed their deadlines outside their loss-tolerance window. In another set of experiments, we ran the same workload on the same setup. Halfway through the experiment, we introduce perturbation in one of the servlet servers by running four linpack[6] processes. The average throughput of the two classes fell to 118 and 115 responses/sec. respectively (Figure 23(b).)

---

[4] A class is a set of request with same requirements

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

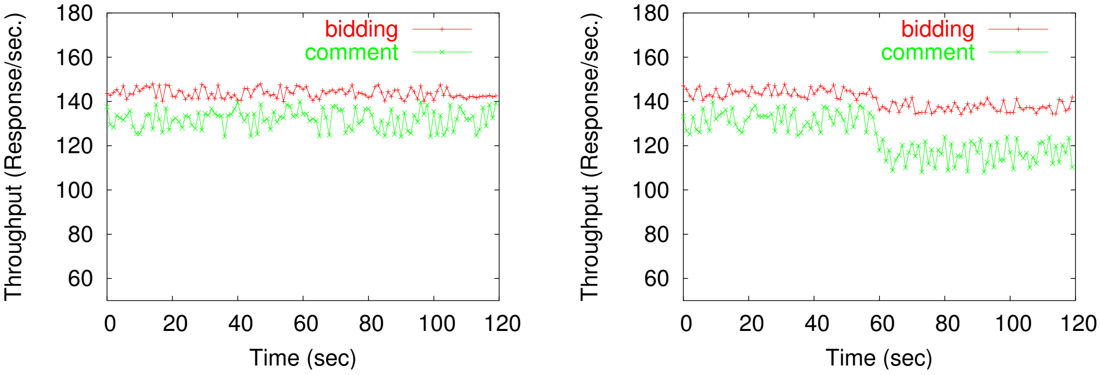
[6] http://www.netlib.org/linpack/



(a) No perturbation (b) With perturbation

Figure 23: Performance throughput with Dynamic Window Constraint Scheduling (DWCS) Algorithm

The drop in the performance and the failure to achieve service differentiation was mainly because of inability of DWCS to take the resource usage and availability into account while making its scheduling and dispatching decisions. An implicit assumption in DWCS is that the capacity of the backend is static. The only observable metrics to DWCS are response time and throughput, which are useful, but not sufficient to determine the resources that are in contention. This is important because different kinds of requests consume different kinds of resources and the dispatching algorithm needs to know exactly what resource is needed to process a request and where is it available. Also, service capacities may vary and that may require change in the number of outstanding responses allowed for that service in order to avoid poor utilization. Similarly, the scheduling algorithm can use the resource usage information of each request class and independently schedule the ones which use complementary resources. Instead of processing a request partially and discarding it because its deadline was not met, the admission control mechanisms can predict whether a particular deadline can be met by looking at the resource availability information. A request will be dropped early in the path if its deadline can't be met and valuable server resources will be saved which can be used to process other queries. In a dynamic heterogeneous environment where workloads and their requirement may change and services availability and capacity



(a) No perturbation

(b) With perturbation

Figure 24: Performance throughput with Resource-Aware Dynamic Window Constraint Scheduling (RA-DWCS) Algorithm

may vary, it's important that the resource-control mechanisms are aware of resources that they are trying to control. Towards that end, we design a simple resource-aware DWCS (RA-DWCS) that use SysProf information to guide its scheduling and dispatching decisions.

Resource-Aware DWCS (RA-DWCS): RA-DWCS adds the following features to the ordinary DWCS:

- *Admission Control*: RA-DWCS consult SysProf to decide whether a particular request can meet its deadline. It keeps track of the past histories of resource consumption of the request class it execute and compares it with resource availability. If the deadline can't be met, the request is dropped.
- *Capacity Planning*: Server capacity may change because of the addition of new software or hardware or a new workload may be introduced which has a completely new resource usage pattern. SysProf's data is used to identify these changes and change the scheduling parameter (like the depth of output queue) automatically.
- *Request Dispatching*: Dispatching decisions are based on per request resource usage and resource availability (on remote servers) information from SysProf.

We apply RA-DWCS to the RUBiS setup (shown in Figure 22) and subject it to the same workload as in the previous experiment with SysProf enabled. Global Performance

Analyzer (GPA) ran on a machine identical to the client machine. Figure 24(a) shows the results of the new algorithm. The average throughput achieved for *bidding* requests is 144 responses/sec. and that of *comment* requests is 132 responses/sec. There is a slight drop (< 2%) in the average throughput which may be due to the overhead of SysProf. Figure 24(b) shows the results of the experiment in which an artificial load of eight linpack processes is started in one of the servlet servers. The *bidding* and *comment* throughput comes down to 138 and 116 respectively. Thus the degradation in throughput is far less as compared to our earlier experiment. It should also be noted that the higher priority *bidding* request has very insignificant drop in performance and this was basically because of the fact that these requests were routed to the server that was lightly loaded. A number of *comment* requests were dropped by RA-DWCS because of the lack of resources. This increased the utilization of the overall infrastructure.

3.5 Related work

Performance monitoring of distributed systems is a frequent topic of investigation, but most solutions choose to operate at some pre-determined level of granularity, with consequent trade-offs between the quality of the information monitored and the associated monitoring overheads. The systems most similar to ours in terms of monitoring are Dproc [6, 7], ganglia [63], supermon [85], Remos [58], MAGNeT [39] and some others. The differences between those systems and SysProf is that we can monitor and track resource usage both at multiple granularities and across multiple machines, and then analyze the resulting information in a hierarchical manner. The outcome is a low overhead monitoring solution.

The notion of request-based analysis is not a new one. It has been used in a number of research projects. Magpie [19] derives the causal paths and resource consumption from application, middleware, and system traces. Pinpoint [30] instruments J2EE middleware to propagate a unique id with each request, and then uses the generated traces to localize faults. In comparison, SysProf does not modify user-level code or instrument data packets. By doing so, we lose some causality information, but the resulting, low overheads allow us to perform online information analysis. Aguilera et al. [9] treat each system as a black box and infer the causal pattern from the passive message traces. However, this approach cannot attribute resource usage correctly because of the absence of internal system information.

Cohen et al. [32] and Strider [91] use various statistical analyses to detect performance problems and system misconfiguration. These analysis can be built into SysProf's GPA to generate metrics that correlate better with high level system behaviors. Causeway [28] and SDI (Stateful Distributed Interposition) [80] provide operating system constructs that the application can use to track its activities in a multi-tiered system. The advantage is that it is possible to do very deterministic analysis with this approach. SysProf, on the other hand, tries to infer the application behaviour automatically and generates information at different level of granularity.

Many kernel instrumentation techniques have been proposed in the literature. Though the focus of our work is not in designing new instrumentation methods, they are still crucial to the basic performance of SysProf. SysProf uses LTT-like [98] methods of generating events. Finally, there are other tools like Dprobes [66], Dtrace [23], and Kerninst [86] that allow dynamic instrumentation of kernel code. Dynamic instrumentation may be desirable in cases where the system has to be debugged, but can't be shut down to apply static instrumentation patches.

Tipme [38] monitors and diagnoses unusually long latencies in an interactive environment on a single machine. ETE [49] requires application level instrumentation to generate end-to-end response times. In comparison, SysProf does not require changes to user-level code, and it can measure latencies and resource consumption on multiple machines. It analyzes performance data in a hierarchical fashion and provides a customizable interface that can be easily tuned at run-time.

3.6 Summary

This chapter describes a toolkit called SysProf that can monitor and analyze different activities in a distributed system at a different level of granularity. The kernel is instrumented to generate performance events that are processed, first by the local analyzers (in their per-CPU buffers) and then by the global analyzers. The toolkit is configurable and permit run-time extensions to add new analysis. The use of performance gears like the selective monitoring, hierarchical analysis, per-CPU buffers, kernel-level messaging and others keep the overhead low. However, certain activities (like the interleaved request) cannot be monitored efficiently without domain-specific knowledge. The toolkit was demonstrated to be useful in detecting performance bottleneck in a shared NFS service and in providing real-time guarantees in an enterprise-based web service. The new resource-aware algorithm was more robust to changes and achieved ∼14% more throughput than a black-box algorithm.

Management of complex applications and IT infrastructures is becoming a key issue in the enterprise domain. Being able to automate the system and reduce human intervention can increase efficiency, reduce errors and significantly cut down IT costs. The next generation enterprise applications will be evaluated more on the basis of the ability to achieve QoS goals, Service Level Agreements and business revenue generated than on overall raw performance. This requires the system to be able to constantly monitor and analyze the services that are offered to the clients and give feedback to them, thereby forming a closed-loop system that can constantly adapt and tune itself to the changing workload, resources and business demand. The cumulative benefits in terms of decreased management complexity and higher quality of service easily offsets the cost due to monitoring overhead and with the recent trends in the hardware towards multi-core platform, it won't be unusual to have a core dedicated to the analysis of the services that run on that platform.

CHAPTER IV

QMON: QOS- AND UTILITY-AWARE MONITORING IN ENTERPRISE SYSTEMS

Speed, Quality, and Price, pick any two. - James M. Wallace

4.1 Introduction

Modern enterprises are characterized by growing dynamism, heterogeneity, complexity, and scale. Previous chapters showed how the E2EProf and the SysProf toolkit can extract dependency and resource usage information for online service path behaviour understanding. These online services may concern specific subsystems (e.g., database backend [59, 34] or carry out general tasks such as job scheduling [48], resource allocation [79], service morphing [3, 4, 78] and problem diagnosis [32]. 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 [21] 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 timewise 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 [75, 88] 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. Similarly, in the research domain, while monitoring systems have known and dealt with the importance of minimizing the manner or degree in which they perturb the systems and applications they watch [45], their online use for tasks like program steering [46], for example, has focused on minimizing delay, perturbation, or both, rather than explicitly controlling or managing metrics like these. 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 [97]. 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 [31, 90]), or they provide limited methods for configuring or self-configuring monitoring actions [6, 7].

This chapter 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 extend the QoS capabilities of our E2EProf and SysProf toolkit. 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.

4.2 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 [95]. Figure 25 shows the utility achieved from an application server of the Airline Reservation System run by one of our industrial

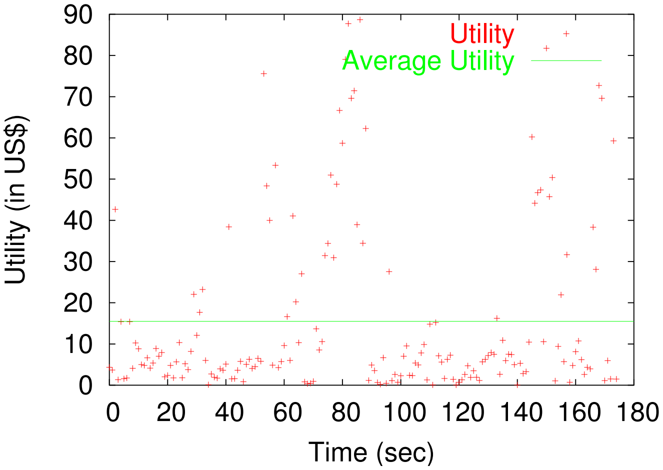


Figure 25: Utility obtained from a server of our partner's Airline Reservation System

partners. In this system, utility($\bigcup$) is defined as follows:

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

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 [20]. 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.

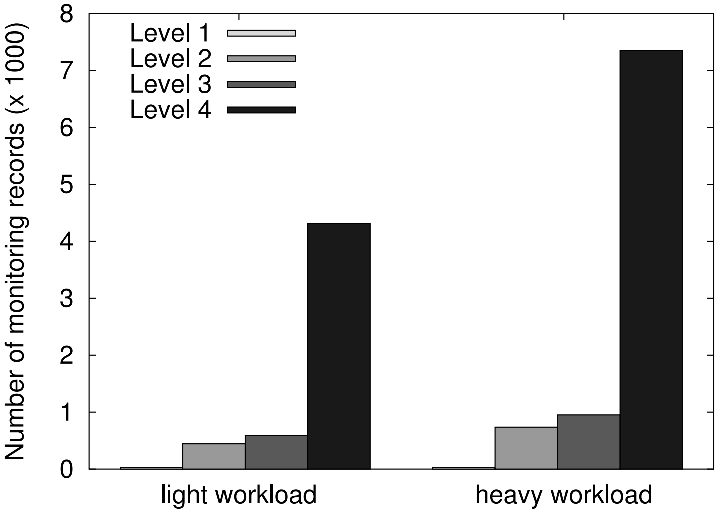


Figure 26: Number of monitoring records per minute

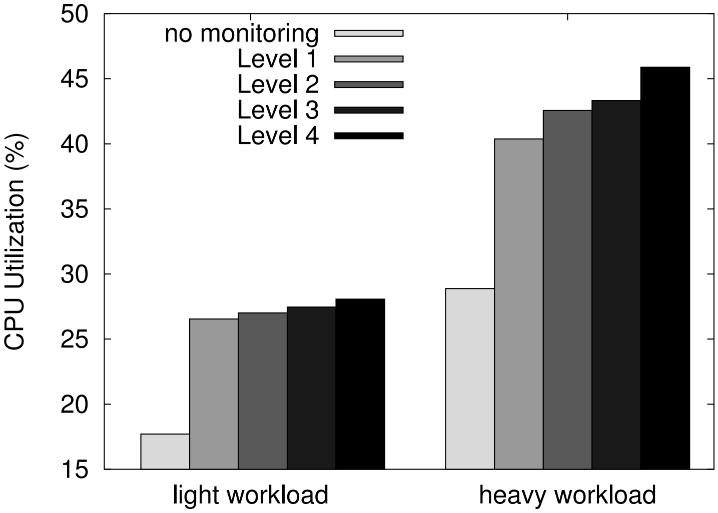


Figure 27: CPU usage in a RUBiS server with different levels of monitoring

A concrete example of the overheads associated with monitoring appears in Figure 26, which shows the number of monitoring messages generated by the OVTA monitoring system in a sample installation of RUBiS. HP OpenView Transaction Analyzer (OVTA) [75] is a widely used commercial product based on the ARM specification [14], 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 [26]. It implements the core functionalities of an auction site like selling, browsing and bidding. Figure 26 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 unresponsive, and fail to meet users' monitoring QoS requirements. In the case of OVTA, this limit is 7200 records/minute [75]. Furthermore, an increased level of monitoring also implies additional overhead at the application servers being monitored. For example, Figure 27 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 chapter 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 ($\bigcup$) or revenue earned by an IT service is usually related to

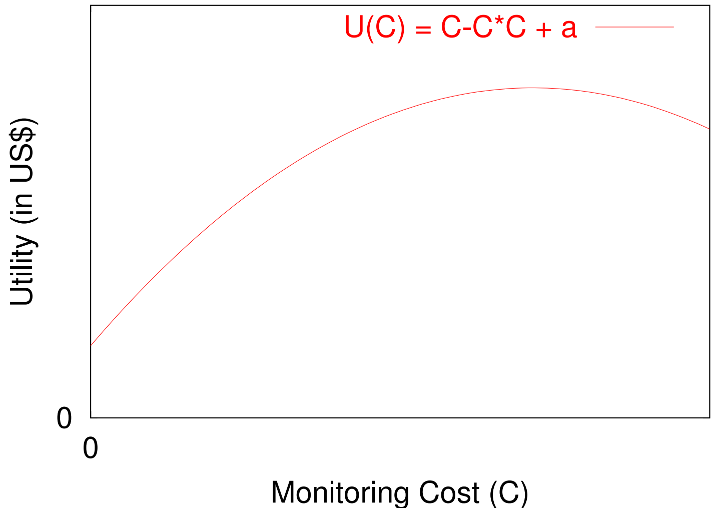


Figure 28: Utility - Monitoring Cost Relationship

the monitoring cost (C) as follows:

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

Figure 28 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 4.4, 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.

4.3 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* can be controlled by a policy engine driving the monitoring system's management [84].

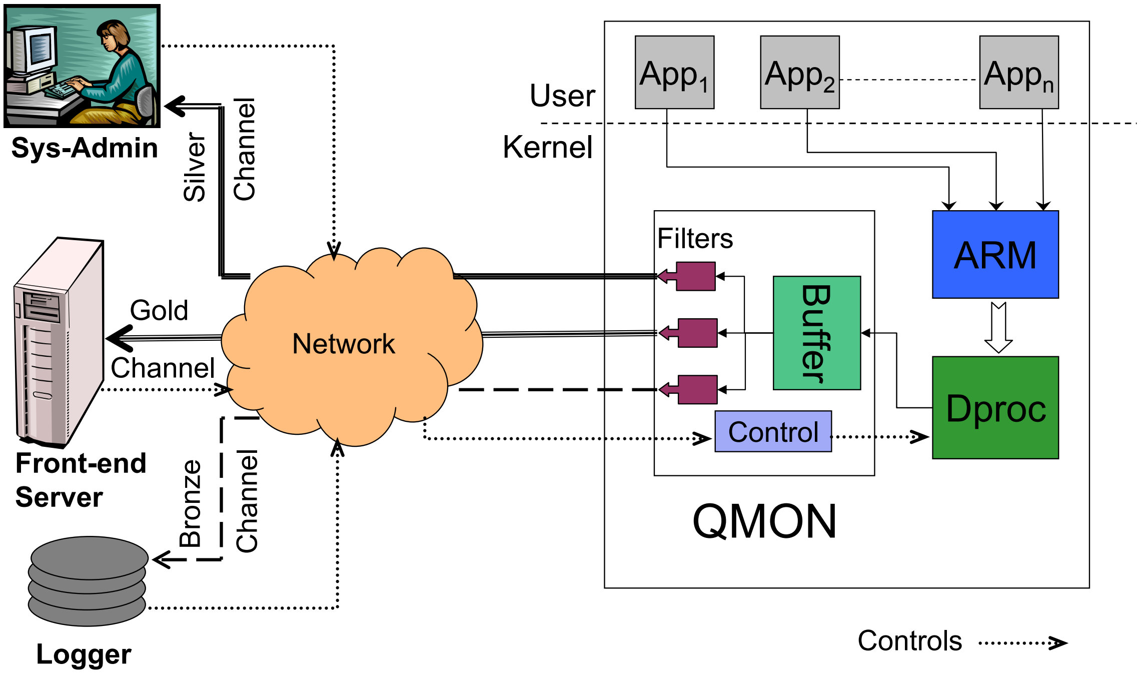


Figure 29: QMON Architecture

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

- *What to monitor* (i.e., which resources, services)?
- *How much to monitor* (i.e., level of detail, (aggregate metrics or per process etc.), 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 organize monitoring overlays? Should the above processing be done at the source or at intermediate nodes?

- *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 29). The channel notion is based on our earlier work on publish-subscribe middleware [37], 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) [36] 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*' [6, 7]. *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 [14] 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.

```
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;
}
```

Figure 30: ARM Filter E-Code

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 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 30 shows an example configuration of ARM data and a *dproc* filter that calculates the average time taken by different types of transactions.

4.4 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 enterprise 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 30 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.

4.4.1 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.

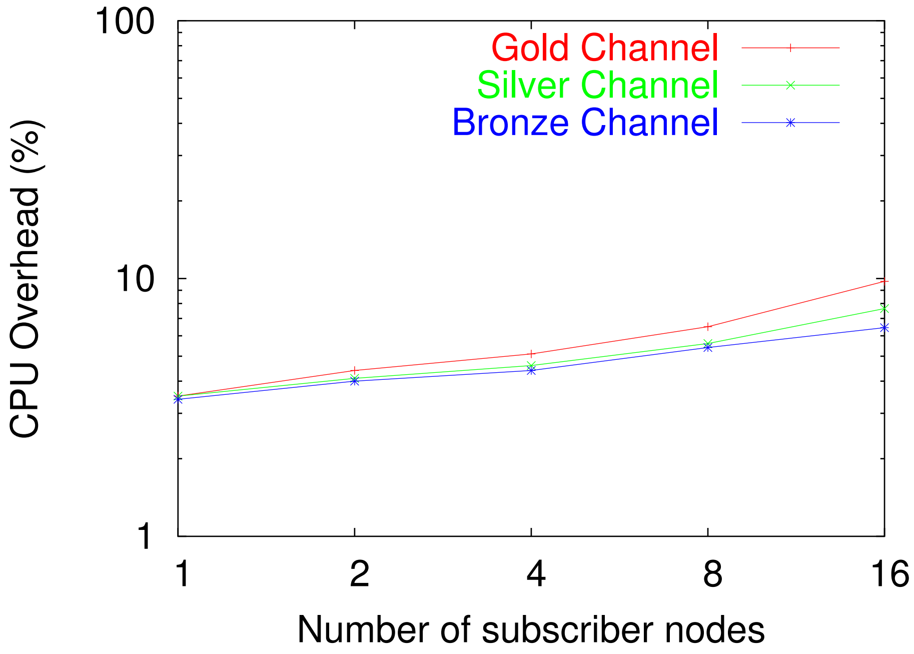


Figure 31: Microbenchmark: Publisher CPU Overhead

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 31 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.

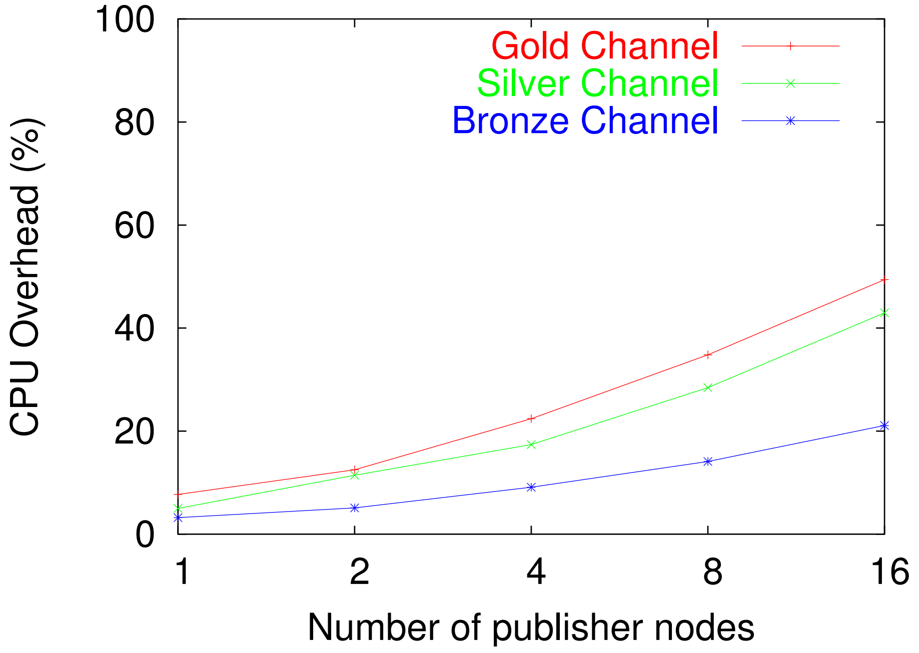


Figure 32: Microbenchmark: Subscriber CPU Overhead

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 31), the cost at the analyzing node increases rapidly with the increase in the number of publishers (see Figure 32). This is where QMON's flexibility 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.

4.4.2 Application Benchmark

As discussed earlier, we employ RUBiS to illustrate the efficacy of QMON. RUBiS is available in three different flavors: PHP, Java HTTP Servlets, and EJB. We use the Servlets version of RUBiS with Apache 2.0.40 web server at the front end, two Jakarta Tomcat 5.5.9 servlet 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.

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 33 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

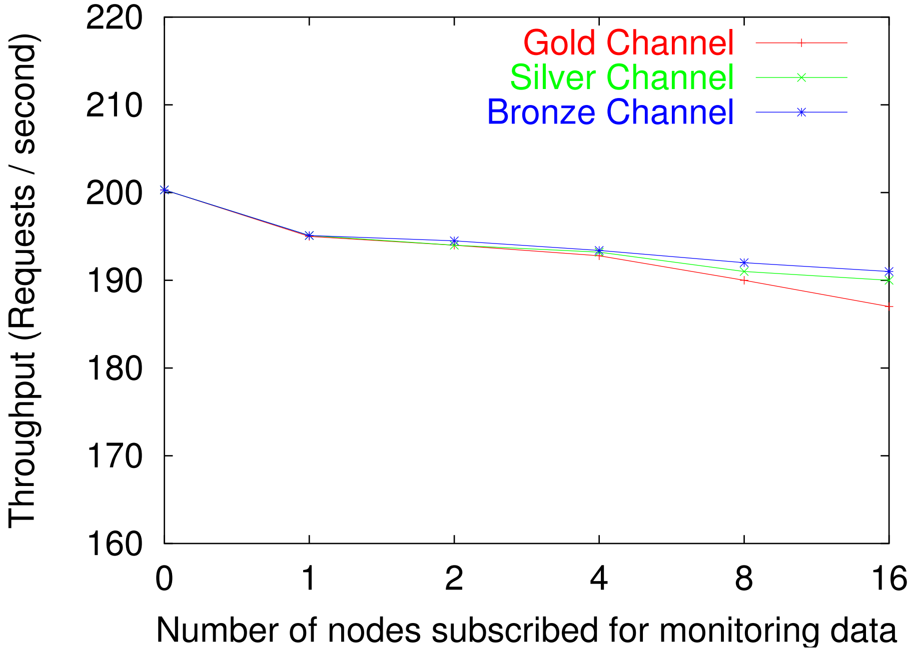


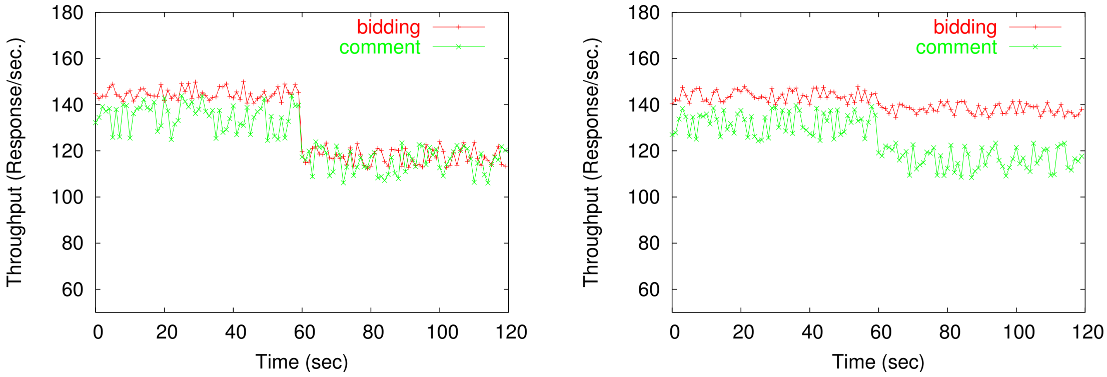
Figure 33: Throughput degradation due to QMON

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* [94]. DWCS was originally developed for streaming multimedia applications, to schedule their processes and/or perform message scheduling [93]. 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 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* [67] 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.



(a) QMON Disabled (b) QMON Enabled

Figure 34: Performance throughput with Application Interference

4.4.2.1 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 reschedule the foreign application to a different machine. This requires monitoring support

---

[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.

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 [61].

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 34(a)). In Figure 34(b), 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.

4.4.2.2 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 [61]. 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 4.2), 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:

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

Figure 35 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%.

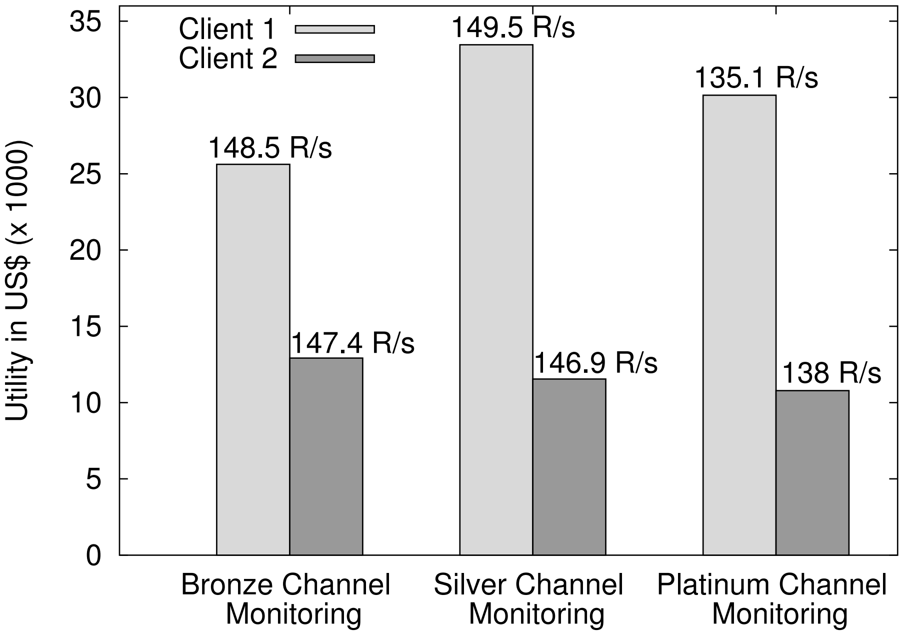


Figure 35: Change in Utility due to QoS in monitoring

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.

4.5 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 4.4.2.1 and 4.4.2.2, 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 4.4.2.2, 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.

4.6 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 [16] and more recently, in the mobile domain [27], 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 [63] is a scalable distributed monitoring system for high performance computing systems. MonALISA [68] provides a distributed monitoring service based on a scalable dynamic distributed architecture. ACME [74] 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 [73] (performance agent, transaction analyzer, network node manager, performance manager, etc.) implement a flexible distributed monitoring solution for enterprise management. IBM's Tivoli monitoring [88] 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.

4.7 Summary

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 chapter 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.

CHAPTER V

RELATED WORK

There has been lot of research in online performance understanding of distributed system behavior recently. Our work is a first attempt to perform online, end-to-end, QoS-aware and non-intrusive monitoring of enterprise systems. Path-based analysis of distributed systems has been studied repeatedly, including in recent research reported in ETE [49]. ETE measures the latencies between component interactions and relates them to end-to-end response times to detect performance problems. Their only performance metric is latency, which may not be sufficient for understanding complex workload behavior. Photon [89] tagged each outgoing MPI message with some context information and the modified MPI runtime at receiver side generate latency statistics from these Aguilera et al. [9] presented a statistical method of determining causally dependent paths and the latency involved in a distributed systems, with tracing support that added little additional overhead. However, they cannot detect the rare anomalies and the exact causes of problems in request paths.

Pinpoint [29] detects system components where requests fail, by tagging (and propagating) a unique request ID with each request. E2Eprof does not add such message overheads and in addition, it also keeps track of resource consumption and other statistics. Another system that does request tracking is Magpie [19]. Magpie requires no global ID, and it can keep a trace of resource consumption like disk accesses, latency, and so on. E2Eprof collects performance profiles at a finer granularity, e.g., by making use of hardware counters, which is unique compared to any of the other systems reviewed herein. Furthermore, E2Eprof requires no changes to user-level applications, thereby creating general and easily used methods for profiling the behavior of distributed programs.

Many techniques have been proposed for monitoring the low-level performance of systems. Compiler-level instrumentation is commonly used to understand program behaviors (e.g. gprof [44]). However, source code may not always be available, and the sizes and complexities of sources are disincentives for software engineers engaged in post-development instrumentation or evaluation. Binary instrumentation is an option, but it requires some level of understanding of application details.

Tracing tools for single systems like the Linux Trace Toolkit [98] and Dtrace [23] provide mechanisms for logging events by inserting instrumentation code. Ktracer shares with these tools its basic ideas of writing logs without blocking, user-specified filtering, etc.

EtE [41] and Certes [72] measure client-perceived response time at the server side. The former does offline analysis of the packets sent and received at the server side, while the latter does online analysis by observing the states of TCP connections. SysProf can be used to obtain more detailed analyses, by pointing out different bottlenecks at the server side.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

This thesis addresses the important and growing class of large-scale distributed applications that offer end-to-end guarantees for the services they provide. We have developed automated methods for capturing the *service paths* traversed by requests or messages and performing runtime analyses like critical path discovery and dynamic bottleneck detection. Supported by compact representations of dynamic path behaviors and properties, these methods detect the occurrence of problems in service paths and pinpoint their locations, thereby facilitating the tasks of human operators or end users.

The thesis makes three major technical contributions. The *E2EProf* toolkit uses an online time-series analysis algorithm called *pathmap* to detect the paths taken by requests and delays incurred due to different path components. It neither requires modification of any deployed components, nor does it make any assumptions about the applications. The *SysProf* toolkit goes a step further in analysing the service paths by instrumenting the operating system kernel to measure the resources consumed by different classes of requests at different physical nodes in distributed applications. No application modification is necessary and therefore, it can handle legacy applications as well. Finally, the *QMON* toolkit provides APIs and communication mechanisms to disseminate monitoring informations generated from E2EProf and SysProf such that the balance between the monitoring cost and utility is achieved automatically. These toolkits has been used extensively to diagnose performance behavior and perform system management tasks in a variety of enterprise and scientific applications.

The service path approach and the monitoring toolkits presented in this thesis provide a simple and non-intrusive way to perform online performance understanding. Future enterprise applications can take advantages of our toolkits and can build specialized services on top of our monitoring facilities and provide automated management capabilities. This thesis also opens several opportunities for future research directions. Technology trends suggest that more and more machines would be using virtualization and multi-core architectures. It would be interesting to study the service path approach and its assumptions in the context of these new architectures.

REFERENCES

[1] Agarwala, S., Alegre, F., Schwan, K., and Mehalingham, J., "E2EProf: Automated End-to-End Performance Management for Enterprise Systems," in *In the Proceedings of 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007), Edinburgh, UK.*, June 2007.

[2] Agarwala, S., Chen, Y., Milojicic, D., and Schwan, K., "QMON: QoS- and Utility-Aware Monitoring in Enterprise Systems," in *The 3rd IEEE International Conference on Autonomic Computing (ICAC 2006), Dublin, Ireland*, pp. 124–133, June 2006.

[3] Agarwala, S., Eisenhauer, G., and Schwan, K., "Morphable Messaging: Efficient Support for Evolution in Distributed Applications," in *2nd International Workshop on Challenges of Large Applications in Distributed Environments (CLADE 2004), Honolulu, HI, USA*, pp. 86–95, June 2004.

[4] Agarwala, S., Eisenhauer, G., and Schwan, K., "Lightweight Morphing Support for Evolving Data Exchanges in Distributed Applications," in *25th International Conference on Distributed Computing Systems (ICDCS 2005)*, pp. 697–706, June 2005.

[5] Agarwala, S., Paul, A., Ramachandran, U., and Schwan, K., "e-SAFE: An Extensible, Secure and Fault Tolerant Storage System," in *In the Proceedings of First IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2007), Boston, MA.*, July 2007.

[6] Agarwala, S., Poellabauer, C., Kong, J., Schwan, K., and Wolf, M., "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*, pp. 250 – 259, June 2003.

[7] Agarwala, S., Poellabauer, C., Kong, J., Schwan, K., and Wolf, M., "System-Level Resource Monitoring in High-Performance Computing Environments," *Journal of Grid Computing*, vol. 1, pp. 273–289, September 2003.

[8] Agarwala, S. and Schwan, K., "SysProf: Online Distributed Behavior Diagnosis through Fine-grain System Monitoring," in *The 26th International Conference on Distributed Computing Systems (ICDCS 2006), Lisboa, Portugal*, p. 8, July 2006.

[9] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A., "Performance debugging for distributed systems of black boxes," in *Proceedings of the 19th ACM Symposium on Operating Systems Principles, Bolton Landing, NY USA*, pp. 74–89, October 2003.

[10] Alonso, G., Casati, F., Kuno, H., and Machiraju, V., *Web Services Concepts, Architectures and Applications.* Springer Verlag, 2004.

[11] Anderson, D. C., Chase, J. S., and Vahdat, A., "Interposed request routing for scalable network storage," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, pp. 25–48, February 2002.

[12] Andreolini, M., Colajanni, M., and Morselli, R., "Performance study of dispatching algorithms in multi-tier web architectures," in *Proceedings of the International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS), Marina Del Rey, California, USA*, pp. 10–20, June 2002.

[13] Arlitt, M. and Jin, T., "A workload characterization study of the 1998 World Cup Web site," *IEEE Network*, vol. 14, pp. 30–37, May 2000.

[14] "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` (Retrieved on May 1st 2007).

[15] Aron, M., Druschel, P., and Zwaenepoel, W., "Cluster reserves: a mechanism for resource management in cluster-based network servers," in *SIGMETRICS*, pp. 90–101, 2000.

[16] Aurrecoechea, C., Campbell, A. T., and Hauw, L., "A survey of QoS architectures," *Multimedia Systems*, vol. 6, pp. 138 – 151, May 1998.

[17] "An architectural blueprint for autonomic computing," April 2003. `http://www-03.ibm.com/autonomic/blueprint.shtml`.

[18] Banga, G., Druschel, P., and Mogul, J. C., "Resource Containers: A New Facility for Resource Management in Server Systems," in *Proceedings of the Third USENIX Symposium on Operating Systems Design and Implementation (OSDI), New Orleans, Louisiana*, pp. 45–58, February 1999.

[19] Barham, P. T., Donnelly, A., Isaacs, R., and Mortier, R., "Using Magpie for Request Extraction and Workload Modelling," in *6th USENIX Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA*, pp. 259–272, December 2004.

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

[21] Bhatti, N. and Friedrich, R., "Web server support for tiered services," *IEEE Network*, vol. 13, pp. 64–71, September 1999.

[22] Blanquer, J. M., Batchelli, A., Schauser, K., and Wolski, R., "Quorum: Flexible Quality of Service for Internet Services," in *2nd USENIX Symposium on Networked Systems Design and Implementation (NSDI '05), Boston, Massachusetts, USA*, May 2005.

[23] Cantrill, B., Shapiro, M. W., and Leventhal, A. H., "Dynamic Instrumentation of Production Systems," in *Proceedings of the General Track: 2004 USENIX Annual Technical Conference, Boston, MA, USA*, pp. 15–28, June 2004.

[24] Cardellini, V., Casalicchio, E., Colajanni, M., and Yu, P. S., "The state of the art in locally distributed Web-server systems," *ACM Computing Survey*, vol. 34, pp. 263–311, June 2002.

[25] "Understanding Common Base Events Specification V1.0.1," July 2003. `http://www-128.ibm.com/developerworks/library/specification/ws-cbe/` (Retrieved on May 1st 2007).

[26] Cecchet, E., Marguerite, J., and Zwaenepoel, W., "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*, pp. 246–261, November 2002.

[27] Chakrabarti, S. and Mishra, A., "QoS issues in ad hoc wireless networks," *Communications Magazine, IEEE*, vol. 39, pp. 142 – 148, February 2001.

[28] Chanda, A., Elmeleegy, K., Cox, A. L., and Zwaenepoel, W., "Causeway: Support for Controlling and Analyzing the Execution of Web-Accessible Applications," in *Middleware*, November 2005.

[29] Chen, M. Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., and Brewer, E., "Path-based failure and evolution management," in *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI), San Francisco, CA*, 2004.

[30] Chen, M. Y., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E., "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pp. 595 – 604, June 2002.

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

[32] Cohen, I., Chase, J. S., Goldszmidt, M., Kelly, T., and Symons, J., "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*, pp. 231–244, December 2004.

[33] Crovella, M. E. and Bestavros, A., "Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes," *IEEE/ACM Transactions on Networking (TON)*, vol. 5, pp. 835–846, December 1997.

[34] Diao, Y., Eskesen, F., Froehlich, S., Hellerstein, J. L., Spainhower, L., and Surendra, M., "Generic 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*, pp. 3–15, October 2003.

[35] Eckerson, W. W., "Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client Server Applications," *Open Information Systems 10, 1*, vol. 3, January 1995.

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

[37] Eisenhauer, G., Schwan, K., and Bustamante, F., "Publish-subscribe for High-performance Computing," *IEEE Computing*, vol. 10, pp. 40–47, January/February 2006.

[38] Endo, Y. and Seltzer, M. I., "Improving interactive performance using TIPME," in *Proceedings of the 2000 ACM SIGMETRICS International Conference on Measurements and Modeling of Computer Systems, Santa Clara, CA*, pp. 240–251, June 2000.

[39] Feng, W., Broxton, M., Engelhart, A., and Hurwitz, G., "MAGNeT: A Tool for Debugging, Analysis and Reflection in Computing Systems," in *Proceedings of Third IEEE International Symposium on Cluster Computing and the Grid, Tokyo, Japan*, pp. 310–317, May 2003.

[40] Foster, I. and Kesselman, C., eds., *The grid: blueprint for a new computing infrastructure.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998.

[41] Fu, Y., Cherkasova, L., Tang, W., and Vahdat, A., "Ete: Passive end-to-end internet service performance monitoring," in *Proceedings of USENIX Annual Technical Conference, Monterey, CA*, June 2002.

[42] Ganev, I. B., Eisenhauer, G., and Schwan, K., "Kernel Plugins: When a VM Is Too Much," in *Proceedings of the 3rd USENIX Virtual Machine Research and Technology Symposium (VM), San Jose, CA, USA*, pp. 83–96, May 2004.

[43] Gavrilovska, A., Schwan, K., and Oleson, V., "A Practical Approach for Zero Downtime in an Operational Information System," in *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pp. 345–354, July 2002.

[44] Graham, S. L., Kessler, P. B., and McKusick, M. K., "gprof: a call graph execution profiler," in *SIGPLAN Symposium on Compiler Construction*, pp. 120–126, 1982.

[45] Gu, W., Eisenhauer, G., Schwan, K., and Vetter, J. S., "Falcon: On-line monitoring for steering parallel programs.," *Concurrency - Practice and Experience*, vol. 10, no. 9, pp. 699–736, 1998.

[46] Gu, W., Vetter, J., and Schwan, K., "An Annotated Bibliography of Interactive Program Steering," *ACM SIGPLAN Notices*, vol. 29, no. 9, pp. 140–148, 1994.

[47] He, Q. and Schwan, K., "Iq-rudp: Coordinating application adaptation with network transport," in *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), Edinburgh, Scotland, UK*, pp. 369–378, July 2002.

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

[49] Hellerstein, J. L., Maccabee, M. M., III, W. N. M., and Turek, J., "ETE: A Customizable Approach to Measuring End-to-End Response Times and Their Components in Distributed Systems," in *Proceedings of the 19th International Conference on Distributed Computing Systems, Austin, TX, USA*, pp. 152–162, June 1999.

[50] Jaiswal, S., Iannaccone, G., Diot, C., Kurose, J., and Towsley, D., "Inferring TCP connection characteristics through passive measurements," in *INFOCOM 2004. Twenty-third Annual Joint Conference of the IEEE Computer and Communications Societies*, pp. 1582–1592, March 2004.

[51] Jiang, H. and Dovrolis, C., "Why is the internet traffic bursty in short time scales?," in *Proceedings of the 2005 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 241–252, June 2005.

[52] jie Jin, L., Machiraju, V., and Sahai, A., "Analysis on Service Level Agreement of Web Services," Tech. Rep. HPL-2002-180, HP Laboratories, Palo Alto, June 2002.

[53] Joe Smart, B. M. and other senior IT managers at Delta Airlines, "." personal communication, 2006. .

[54] Karlsson, M., Karamanolis, C., and Chase, J., "Controllable Fair Queuing for Meeting Performance Goals," in *IFIP International Symposium on Computer Performance Modeling, Measurement and Evaluation (PERFORMANCE), Juan-les-Pins, France*, pp. 278–294, October 2005.

[55] Karlsson, M., Karamanolis, C., and Zhu, X., "Triage: Performance Isolation and Differentiation for Storage Systems," in *the Proceedings of the International Workshop on Quality of Service (IWQoS 2004), Montreal, Canada*, pp. 67–74, June 2004.

[56] Kumar, V., Cai, Z., Cooper, B. F., Schwan, G. E. K., Mansour, M., Seshasayee, B., and Widener, P., "Implementing Diverse Messaging Models with Self-Managing Properties using IFLOW," in *The 3rd IEEE International Conference on Autonomic Computing (ICAC 2006), Dublin, Ireland*, pp. 243–252, June 2006.

[57] Lee, E. K. and Thekkath, C. A., "Petal: Distributed Virtual Disks," in *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems, Cambridge, Massachusetts*, pp. 84–92, October 1996.

[58] Lowekamp, B., Miller, N., Karrer, R., Gross, T., and Steenkiste, P., "Design, Implementation, and Evaluation of the Remos Network Monitoring System," *Journal of Grid Computing*, vol. 1, no. 1, pp. 75–93, 2003.

[59] Lumb, C. R., Merchant, A., and Alvarez, G. A., "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.

[60] Malony, A. D., Reed, D. A., and Wijshoff, H. A. G., "Performance Measurement Intrusion and Perturbation AnalysiPerformance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 3, pp. 433–450, July 1992.

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

[62] Mansour, M. S., Scwhan, K., and Abdelaziz, S., "I-Queue: Smart queues for service management," in *Proceedings of the 4th International Conference on Service OrientedComputing (ICSOC 06)*, Lecture Notes in Computer Science, (Chicago, USA), Springer, 2006.

[63] Massie, M. L., Chun, B. N., and Culler, D. E., "The ganglia distributed monitoring system: Design, implementation, and experience," *Parallel Computing*, vol. 30, pp. 817–840, July 2004.

[64] Mills, D. L., "The network computer as precision timekeeper," in *Proceedings of the Precision Time and Time Interval (PTTI) Applications and Planning Meeting, Reston VA*, pp. 96–108, December 1996.

[65] Mogul, J. C., "Operating Systems Should Support Business Change," in *Tenth Workshop on Hot Topics in Operating Systems (HotOS X), Santa Fe, NM*, July 2005.

[66] Moore, R. J., "A Universal Dynamic Trace for Linux and Other Operating Systems," in *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference, Boston, Massachusetts, USA*, pp. 297–308, June 2001.

[67] Mosberger, D. and Jin, T., "httperf: A Tool for Measuring Web Server Performance," *Performance Evaluation Review*, vol. 26, pp. 31–37, December 1998. Originally appeared in Proceedings of the 1998 Internet Server Performance Workshop, June 1998, 59-67.

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

[69] NSF, "High end computing university research activity (hecura)." `http://www.nsf.gov/pubs/2006/nsf06503/nsf06503.htm` (Retrieved on May 1st 2007).

[70] Oldfield, R. A., Maccabe, A. B., Arunagiri, S., Kordenbrock, T., Riesen, R., Ward, L., and Widener, P., "Lightweight i/o for scientific applications," in *In Proceedings of the 2006 IEEE Conference on Cluster Computing*, September 2006.

[71] Oleson, V., Schwan, K., Eisenhauer, G., Plale, B., Pu, C., and Amin, D., "Operational information systems - an example from the airline industry," in *First Workshop on Industrial Experiences with Systems Software (WIESS)*, October 2000.

[72] Olshefski, D. P., Nieh, J., and Agrawal, D., "Inferring client response time at the web server," in *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pp. 160–171, 2002.

[73] "HP OpenView." `http://www.openview.hp.com` (Retrieved on May 1st 2007).

[74] Oppenheimer, D., Vatkovskiy, V., Weatherspoon, H., Lee, J., Patterson, D. A., and Kubiatowicz, J., "Monitoring, Analyzing, and Controlling Internet-scale

Systems with ACME," Tech. Rep. UCB/CSD-03-1276, EECS Department, University of California, Berkeley, 2004.

[75] "HP OpenView Transaction Analyzer performance and scalability Guide." `http://www.managementsoftware.hp.com/products/tran/` (Retrieved on May 1st 2007).

[76] Paul, A., Agarwala, S., and Ramachandran, U., "e-SAFE: An Extensible, Secure and Fault Tolerant Storage System," Tech. Rep. GIT-CERCS-05-03, CERCS Technical Report, Georgia Institute of Technology, May 2003.

[77] Petrini, F., Kerbyson, D. J., and Pakin, S., "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q," in *Proceedings of the ACM/IEEE SC2003 Conference on High Performance Networking and Computing, (Supercomputing 2003), Phoenix, AZ, USA*, November 2003.

[78] Poellabauer, C., Schwan, K., Agarwala, S., Gavrilovska, A., Eisenhauer, G., Pande, S., Pu, C., and Wolf, M., "Service Morphing: Integrated System- and Application-Level Service Adaptation in Autonomic Systems," in *Proceedings of the 5th Annual International Workshop on Active Middleware Services (AMS)*, June 2003.

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

[80] Reumann, J. and Shin, K. G., "Stateful distributed interposition," *ACM Transactions on Computer Systems (TOCS)*, vol. 22, pp. 1–48, February 2004.

[81] Reynolds, P., Wiener, J. L., Mogul, J. C., Aguilera, M. K., and Vahdat, A., "WAP5: black-box performance debugging for wide-area systems," in *Proceedings of the 15th international conference on World Wide Web, WWW 2006, Edinburgh, Scotland, UK*, pp. 347–356, May 2006.

[82] Schwan, K., Cooper, B. F., Eisenhauer, G., Gavrilovska, A., Wolf, M., Abbasi, H., Agarwala, S., Cai, Z., Kumar, V., Lofstead, J., Mansour, M., Seshasayee, B., and Widener, P., "Autonomic Information Flows," Tech. Rep. GIT-CERCS-05-22, CERCS Technical Report, Georgia Institute of Technology, November 2005.

[83] Schwan, K., Cooper, B. F., Eisenhauer, G., Gavrilovska, A., Wolf, M., Abbasi, H., Agarwala, S., Cai, Z., Kumar, V., Lofstead, J., Mansour, M., Seshasayee, B., and Widener, P., "AutoFlow: Autonomic Information Flows for Critical Information Systems," in *Autonomic Computing: Concepts, Infrastructure, and Applications* (Parashar, M. and Hariri, S., eds.), ch. 14, CRC Press, ISBN# 0849393671, December 2006.

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

[85] Sottile, M. and Minnich, R., "Supermon: A High-Speed Cluster Monitoring System," in *Proceedings of IEEE International Conference on Cluster Computing, Chicago, IL, USA*, pp. 39–46, September 2002.

[86] Tamches, A. and Miller, B. P., "Fine-grained dynamic instrumentation of commodity operating system kernels," in *Proceedings of the Third Symposium on Operating Systems Design and Implementation*, pp. 117–130, February 1999.

[87] Thereska, E., Salmon, B., Strunk, J. D., Wachs, M., Abd-El-Malek, M., Lopez, J., and Ganger, G. R., "Stardust: tracking activity in a distributed storage system," in *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS/Performance, Saint Malo, France*, pp. 3–14, June 2006.

[88] "IBM Tivoli Monitoring.." `http://www-306.ibm.com/software/tivoli/products/monitor/` (Retrieved on May 1st 2007).

[89] Vetter, J., "Dynamic statistical profiling of communication activity in distributed applications," in *Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, Marina Del Rey, CA*, pp. 240 – 250, June 2002.

[90] Vetter, J. S. and Reed, D. A., "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.

[91] Wang, Y.-M., Verbowski, C., Dunagan, J., Chen, Y., Wang, H. J., Yuan, C., and Zhang, Z., "STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support," in *Proceedings of the 17th USENIX Conference on Systems Administration (LISA 2003), San Diego, California, USA*, pp. 159–172, October 2003.

[92] "IT responsiveness and efficiency with IBM WebSphere Extended Deployment," November 2004. `ftp://ftp.software.ibm.com/software/webserver/appserv/library/WS_XD_G22%4-9126-00_WP_Final.pdf` (Retrieved on May 1st 2007).

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

[94] West, R., Zhang, Y., Schwan, K., and Poellabauer, C., "Dynamic Window-Constrained Scheduling of Real-Time Streams in Media Servers," *IEEE Transactions on Computers*, vol. 53, pp. 744–759, June 2004.

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

[96] Wolf, M., Cai, Z., Huang, W., and Schwan, K., "SmartPointers: Personalized Scientific Data Portals in your Hand," in *Proceedings of the 2002 ACM/IEEE conference on Supercomputing, Baltimore, Maryland, USA. Conference on High Performance Networking and Computing*, pp. 1–16, November 2002.

[97] Wolski, R., Spring, N., and Hayes, J., "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," *Journal of Future Generation Computing Systems*, vol. 15, pp. 757–768, October 1999.

[98] Yaghmour, K. and Dagenais, M., "Measuring and Characterizing System Behavior Using Kernel-Level Event Logging," in *Proceedings of the General Track: 2000 USENIX Annual Technical Conference, San Diego, CA, USA*, pp. 13–26, June 2000.

[99] Zhang, W., "Linux Virtual Server for Scalable Network Services," in *Ottawa Linux Symposium*, July 2000.

VITA

Sandip Agarwala is a PhD candidate in the College of Computing at the Georgia Institute of Technology. He received his Bachelor of Technology degree in Computer Science and Engineering from Indian Institute of Technology, Kharagpur in 2001. His research interests are in the general area of experimental computer systems, with primary focus on the design, development and analysis of system- and middleware-level techniques to diagnose performance, manage resources and automate the management of large-scale distributed systems.