MONITORING-AS-A-SERVICE IN THE CLOUD

A Thesis
Presented to
The Academic Faculty

by

Shicong Meng

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology
May 2012

MONITORING-AS-A-SERVICE IN THE CLOUD

To Tina, Mom and Dad.

ACKNOWLEDGEMENTS

I gratefully acknowledge the support and encouragement provided by my advisor, Professor Ling Liu, during the time I spent at Georgia Tech. I feel extremely fortunate to have crossed paths with her and to have had the opportunity to work with her so closely for many years. She has been a most valued friend and mentor during tough times. She has cultivated me with her fantastic taste of research problems while allowing me the freedom of pursuing the problems I found most interesting; at the same time, in critical moments, she has always been available to provide sincere and determined opinions, helping me make the right decisions.

I would like to acknowledge the feedback received from members of my thesis committee: Professor Calton Pu, Professor Karsten Schwan, Professor Leo Mark, Professor Lakshmish Ramaswamy and Professor Francisco Hernandez. I would like to thank them for interesting discussions and inspiring suggestions on my research which helped me look at my work in a much broader context. Special thanks are also due to Professor Calton Pu and Professor Karsten Schwan for thought-provoking conversations on distributed system and Cloud Computing research. Their rapid ideas, unusual associations and amazing ability of finding connections between seemingly disparate problems have always been inspirational for me.

The friendship, companionship and support of my colleagues in the DiSL research group, Systems lab and Databases lab would be hard to replace. A big thanks to all my collaborators at IBM T.J. Watson Research Center and VMware. Special thanks are due to Arun Iyengar, Isabelle Rouvellou, Chitra Venkatramani at T.J. Watson Research Center and Ravi Soundararajan at VMware.

Finally, I would like to dedicate this work to my wife and my parents, whose love and patience have been my constant source of inspiration, without which this accomplishment would have been impossible. My success is also theirs.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

State monitoring is a fundamental building block for Cloud services. The demand for providing state monitoring as services (MaaS) continues to grow and is evidenced by CloudWatch from Amazon EC2, which allows cloud consumers to pay for monitoring a selection of performance metrics with coarse-grained periodical sampling of runtime states. One of the key challenges for wide deployment of MaaS is to provide better balance among a set of critical quality and performance parameters, such as accuracy, cost, scalability and customizability.

This dissertation research is dedicated to innovative research and development of an elastic framework for providing state monitoring as a service (MaaS). We analyze limitations of existing techniques, systematically identify the need and the challenges at different layers of a Cloud monitoring service platform, and develop a suite of distributed monitoring techniques to support for flexible monitoring infrastructure, cost-effective state monitoring and monitoring-enhanced Cloud management. At the monitoring infrastructure layer, we develop techniques to support multi-tenancy of monitoring services by exploring cost sharing between monitoring tasks and safeguarding monitoring resource usage. To provide elasticity in monitoring, we propose techniques to allow the monitoring infrastructure to self-scale with monitoring demand. At the cost-effective state monitoring layer, we devise several new state monitoring functionalities to meet unique functional requirements in Cloud monitoring. Violation likelihood state monitoring explores the benefits of consolidating monitoring workloads by allowing utility-driven monitoring intensity tuning on individual monitoring tasks and identifying correlations between monitoring tasks. Window based state monitoring leverages distributed windows for the best monitoring accuracy and communication efficiency. Reliable state monitoring is robust to both transient and long-lasting communication issues caused by component failures or cross-VM performance interferences. At the monitoring-enhanced Cloud management layer, we devise a novel technique to learn about the performance characteristics of both Cloud infrastructure and Cloud applications from cumulative performance monitoring data to increase the cloud deployment efficiency.

CHAPTER I

INTRODUCTION

Cloud computing and its pay-as-you-go economic model not only enable application developers and application service providers to perform on-demand utility computing, but also push the evolution of datacenter technologies to become more open and more consumer-driven. Typically, in addition to rent virtual server instances and pay for certain middleware services based on their usage, such as load balancing in EC2, Cloud consumers also need to monitor the performance of their applications in response to unexpected peaks of service requests or performance degradation in their multi-tier application frameworks. Similarly, Cloud providers need to monitor the large number of computing nodes in their datacenters in response to virtual machine failures or performance degradation of virtual machines, ensuring the level of service quality agreement demanded by the Cloud consumers.

Today's Cloud datacenters are complex composition of large-scale servers, virtual machines, physical and virtual networks, middleware, applications, and services. Their growing scale and complexity challenge our ability to closely monitor the state of various entities, and to utilize voluminous monitoring data for better operation. Providing Monitoring-as-a-Service(MaaS) to Cloud administrators and users brings a number of benefits to both Cloud providers and consumers.

First, MaaS minimizes the cost of ownership by leveraging the state of the art monitoring tools and functionalities. MaaS makes it easier for users to deploy state monitoring at different levels of Cloud services compared with developing ad-hoc monitoring tools or setting up dedicated monitoring hardware/software.

Second, MaaS enables the pay-as-you-go utility model for state monitoring. This is especially important for users to enjoy full-featured monitoring services based on their monitoring needs and available budget. Third, MaaS also brings Cloud service providers the opportunity to consolidate monitoring demands at different levels (infrastructure, platform, and application) to achieve efficient and scalable monitoring.

Finally, MaaS pushes Cloud service providers to invest in state of the art monitoring technology and deliver continuous improvements on both monitoring service quality and performance. With the consolidated services and monitoring data, Cloud service providers can also develop value-add services for better Cloud environments and creating new revenue sources.

We conjecture that monitoring-as-a-service paradigms will become dominating trend for on-demand computing in future Cloud datacenters. This dissertation research tackles the emerging research theme of *providing advanced monitoring functionalities as Cloud services to help users to manage Cloud and harness its power*.

1.1 Technical Challenges

Despite the attractiveness of MaaS, providing monitoring-as-a-service also involves big challenges at different levels.

Cloud-scale monitoring infrastructure. MaaS requires a Cloud-scale monitoring infrastructure with strict performance and scalability requirements. How can we collect a massive set of live information from hundreds of thousands of, even millions of manageable instances in a Cloud datacenter? Due to the on-demand provisioning nature of Cloud, monitoring demands can also change significantly over time. Hence, the monitoring infrastructure should not only achieve high scalability, but also embrace changes in monitoring demands. Furthermore, the monitoring infrastructure must also provide good multi-tenancy support to ensure a massive number of users enjoy Cloud monitoring services at the same time.

Advanced monitoring functionalities. Cloud monitoring needs vary heavily from task to task, and many monitoring tasks requires the support of advanced monitoring techniques to achieve communication efficiency, flexible tradeoff between accuracy and sampling cost as well as reliable distributed monitoring. For instance, Cloud service rate limiting requires intensive monitoring of per-user access rates across a large number of distributed servers which may be located in different continents. Such monitoring tasks require highly efficient monitoring-related communication. As another example, some monitoring tasks such as network traffic monitoring incur high monitoring data collection (sampling) cost. Achieving accurate yet efficient monitoring for these tasks is difficult. Furthermore, failures and malfunctions are the norm rather than the exception in large-scale distributed environments. As a result, monitoring data are almost always error-prone or incomplete. How can we prevent such data from generating misleading monitoring results? Or how can we maximize the utility of monitoring data with the presence of possible disruptions from different levels?

Utilization of monitoring data. Cloud datacenter monitoring generates tremendous amounts of data which often yield little usage besides simple event detection. For example, Amazon EC2's monitoring service CloudWatch[1] provides continuous web application performance and resource usage monitoring for simple dynamic server provisioning (auto-scaling), which also produces considerable monitoring data. Can we leverage such data to offer intelligent functionalities to further simplify Cloud usage? For instance, performance-driven Cloud application provisioning is difficult due to the large number of candidate provisioning plans (e.g., different types of VMs, different cluster configurations, different hourly renting cost, etc.). Is it possible to utilize Cloud application performance monitoring data to simplify the provisioning planning process or even liberate Cloud users from the details of application provisioning and meet their performance goal at the same time? If it is possible, what techniques should we develop to support such functionalities?

1.2 Dissertation Scope and Contributions

This dissertation research tackles the above problems with a layered approach that systematically addresses monitoring efficiency, scalability, reliability and utility at the monitoring infrastructure level, the monitoring functionality level and the monitoring data utility level. We analyze key limitations of existing techniques, and develop new techniques to offer more effective Cloud monitoring capabilities in this layered design. In addition, we built systems that help Cloud developers and users to access, process and utilize Cloud monitoring data. Specifically, this dissertation makes the following contributions in order to address the challenges described in the previous section.

1.2.1 Monitoring Infrastructure

At the monitoring infrastructure level, we propose REMO [79, 78] and Tide [81] which contribute to a Cloud-scale monitoring infrastructure that ensures the efficiency, scalability and multi-tenancy support of Cloud monitoring.

Monitoring Topology Planning [79, 78]. Large-scale monitoring can incur significant overhead on distributed nodes participating in collection and processing of monitoring data. Existing techniques that focus on monitoring task level efficiency often introduce heavily skewed workload distributions on monitoring nodes and cause excessive resource usage on certain nodes. We developed REMO, a resource-aware monitoring system that considers node-level resource constraints, e.g. monitoring-related CPU utilization should less than 5%, as the first-class factor for scheduling multiple monitoring tasks collectively. REMO optimizes the throughput of the entire monitoring network without causing excessive resource consumption on any participating node, which ensures performance isolation in multi-tenent monitoring environments. It also explores cost sharing opportunities among tasks to optimize monitoring efficiency. We prototyped REMO on Sysem S, a large-scale distributed stream processing system built at IBM TJ Watson Lab. Through resource-aware planning, REMO achieves 35%-45% error reduction compared to existing techniques.

Self-Scaling Monitoring Infrastructure [81]. From traces collected in production datacenters, we found that monitoring and management workloads in Cloud datacenters tend to be highly volatile due to their on-demand usage model. Such workloads often makes the management server a performance bottleneck. To address this problem, we developed Tide, a self-scaling management system which automatically scales up or down its capacity according to the observed workloads. We built the prototype of Tide by modifying VMware's vSphere management server and leveraging non-SQL Hadoop based HBase for scalable state persistence. The experimental results show that Tide provides consistent performance even with extreme volatile management workloads through self-scaling.

1.2.2 Monitoring Functionalities to Meet Unique Cloud Monitoring Requirements

At the monitoring functionality level, we aim at providing new monitoring techniques to meet the unique and diverse Cloud monitoring needs, and we propose WISE [83, 82], Volley [76] and CrystalBall [80] which deliver accurate, cost-effective and reliable monitoring results by employing novel distributed monitoring algorithms to process error-prone Cloud environments.

Efficient Continuous State Violation Detection [82][83]. Most existing works on distributed state monitoring employ an instantaneous monitoring model, where the state is evaluated based on the most recent collected results, to simplify algorithm design. Such a model, however, tends to introduce false state alerts due to noises and outliers in monitoring data. To address this issue, we proposed WISE, window based state monitoring which utilizes temporal windows to capture continuous state violation in a distributed setting. WISE not only delivers the same results as those of a centralized monitoring system with a distributed implementation, but also decouples a global monitoring task into distributed local ones in a way that minimizes the overall communication cost.

Violation-Likelihood based Monitoring [76]. Asynchronized monitoring techniques such as periodical sampling often introduce cost-accuracy dilemma, e.g., frequent polling state information may produce fine-grained monitoring data but may also introduce high sampling cost for tasks such as deep packet inspection based network monitoring. To address this issue, we proposed Volley, a violation likelihood based approach which dynamically tunes monitoring intensity based on the likelihood of detecting important results. More importantly, it always safeguards a user-specified accuracy goal while minimizing monitoring cost. Volley also coordinates sampling over distributed nodes to maintain the task-level accuracy, and leverages inter-task state correlation to optimize multi-task sampling scheduling. When deployed in a testbed datacenter environment with 800 virtual machines, Volley reduces monitoring overhead up to 90% with negligible accuracy loss.

Fault-Tolerant State Monitoring [80]. While we often assume monitoring results are trustworthy and monitoring services are reliable, such assumptions do not always hold, especially in large scale distributed environments such as datacenters where transient device/network failures are the norm rather than the exception. As a result, distributed state monitoring approaches that depend on reliable communication may produce inaccurate results with the presence of failures. We developed CrystalBall, a robust distributed state monitoring approach that produces reliable monitoring results by continuously updating the accuracy estimation of the current results based on observed failures. It also adapts to long-term failures by coordinating distributed monitoring tasks to minimize accuracy loss caused by failures. Experimental results show that CrystalBall consistently improves monitoring accuracy even under severe message loss and delay.

1.2.3 State Monitoring Enhanced Cloud Management [77]

At the monitoring data utility level, we study intelligent techniques that utilize monitoring data to offer advanced monitoring management capabilities. As an initial attempt, we propose Prism [77] which offers an innovative application provisioning functionality based on knowledge learned from cumulative monitoring data. We aim at utilizing multi-tier Cloud application performance data to guide application provisioning. Prism is a prediction-based provisioning framework that simplifies application provisioning by using performance prediction to find a proper provisioning plan for a performance goal in a huge space of candidate plans. As its unique feature, Prism isolates and captures the performance impact of different provisioning options, e.g., virtual machine types and cluster configurations, from performance monitoring data with off-the-shelf machine learning techniques. This technique avoids exploring the huge space of candidate provisioning plans with experiments. As a result, Prism can quickly find the most cost-effective plan with little cost for training performance prediction models.

1.3 Organization of the Dissertation

This dissertation is organized as a series of chapters each addressing one of the problems described above. Each chapter presents the detail of the problem being addressed, provides basic concepts and then describes the development of a solution followed by the evaluation of the proposed solution. Related work is described along with each chapter. Concretely, the dissertation is organized as follows.

Chapter 2 and Chapter 3 introduce two systems designed to support multi-tenancy and self-scaling of the monitoring infrastructure. In Chapter 2, we present REMO, a REsource-aware application state MOnitoring system, to address the challenge of monitoring overlay construction. REMO distinguishes itself from existing works in several key aspects. First, it jointly considers inter-task cost sharing opportunities and node-level resource constraints. Furthermore, it explicitly models the per-message processing overhead which can be substantial but is often ignored by previous works. Second, REMO produces a forest of optimized monitoring trees through iterations of two phases. One phase explores cost-sharing opportunities between tasks, and the other refines the tree with resource-sensitive construction schemes. Finally, REMO also employs an adaptive algorithm that balances the benefits and costs of overlay adaptation. This is particularly useful for large systems with constantly changing monitoring tasks. Moreover, we enhance REMO in terms of both performance and applicability with a series of optimization and extension techniques. We perform extensive experiments including deploying REMO on a BlueGene/P rack running IBMs large-scale distributed streaming system - System S Using REMO in the context of collecting over 200 monitoring tasks for an application deployed across 200 nodes results in a 35%-45% decrease in the percentage error of collected attributes compared to existing schemes.

In Chapter 3, we study the problem of achieving self-scaling in datacenter management middleware. Enabling self-scaling in management middleware involves two challenges. First, the self-scaling process should take minimum time during workload bursts to avoid task execution delays. Second, it should utilize as few resources as possible to avoid resource contention with application usage. To meet these two goals, we propose Tide, a self-scaling framework for virtualized datacenter management. Tide is a distributed management server that can dynamically self-provision new management instances to meet the demand of management workloads. Tide achieves responsive and efficient self-scaling through a set of novel techniques, including a fast capacity-provisioning algorithm that supplies just-enough capacity and a workload dispatching scheme that maximizes task execution throughput with optimized task assignment. We evaluate the effectiveness of Tide with both synthetic and real world datacenter management traces. The results indicate that Tide significantly reduces the task execution delay for bursty management workloads. Furthermore, it also minimizes the number of dynamically provisioned management instances by fully utilizing provisioned instances.

Chapter 4, Chapter 5 and Chapter 6 describe techniques that offer unique Cloud monitoring capabilities to meet highly diverse Cloud monitoring needs. In Chapter 4, we present a WIndow-based StatE monitoring framework (WISE) for efficiently managing applications in Cloud datacenters. Window-based state monitoring reports alerts only when state violation is continuous within a specified time window. Our formal analysis and experimental evaluation of WISE both demonstrate that window-based state monitoring is not only more resilient to temporary value bursts and outliers, but also can save considerable communication when implemented in a distributed manner. Experimental results show that WISE reduces communication by 50%-90% compared with instantaneous monitoring approaches and simple alternative schemes.

In Chapter 5, we aim at addressing this problem by presenting Volley, a violation likelihood based approach for efficient distributed state monitoring in datacenter environments. Volley achieves both efficiency and accuracy with a flexible monitoring framework which uses dynamic monitoring intervals determined by the likelihood of detecting state violations. Our approach consists of three techniques. First, we devise efficient node-level adaptation algorithms that minimize monitoring cost with controlled accuracy for both basic and advanced state monitoring models. Second, Volley employs a distributed scheme that coordinates the monitoring on multiple monitoring nodes of the same task for optimal monitoring efficiency. Finally, Volley enables cost reduction with minimum accuracy loss by exploring state correlation at the multi-task level, which is important for addressing workload issues in large-scale datacenters. We perform extensive experiments to evaluate our approach on a testbed Cloud datacenter environment consisting of 800 VMs. Our results on system, network and application level monitoring show that Volley can reduce considerable monitoring cost and still deliver user specified monitoring accuracy under various monitoring scenarios.

In Chapter 6, we introduce a new state monitoring approach that addresses this issue by exposing and handling communication dynamics such as message delay and loss in Cloud monitoring environments. Our approach delivers two distinct features. First, it quantitatively estimates the accuracy of monitoring outputs to capture uncertainties introduced by messaging dynamics. This feature helps users to distinguish trustworthy monitoring results from ones heavily deviated from the truth, and is important for large-scale distributed monitoring where temporary communication issues are common. Second, our approach also adapts to non-transient messaging issues by reconfiguring distributed monitoring algorithms to minimize monitoring errors. Our experimental results show that, even under severe message loss and delay, our approach consistently improves monitoring accuracy, and when applied to Cloud application auto-scaling, outperforms existing state monitoring techniques in terms of the ability to correctly trigger dynamic provisioning.

Chapter 7 presents Prism, a provisioning planning method which finds the most cost-effective provisioning plan for a given performance goal by searching the space of candidate plans with performance prediction. Prism employs a set of novel techniques that can efficiently learn performance traits of applications, virtual machines and clusters from cumulative monitoring data to build models to predict the performance for an arbitrary provisioning plan. It utilizes historical performance monitoring data and data collected from a small set of automatic experiments to build a composite performance prediction model that takes application workloads, types of virtual server instances and cluster configuration as input, and outputs predicted performance.

In Chapter 8, We conclude this dissertation with an overview of contributions of this dissertation research. We also discuss open problems and potential future research directions.

CHAPTER II

RESOURCE-AWARE APPLICATION STATE MONITORING

2.1 Introduction

Recently, we have witnessed a fast growing set of large-scale distributed applications ranging from stream processing [53] to applications [48] running in Cloud datacenters. Correspondingly, the demand for monitoring the functioning of these applications also increases substantially. Typical monitoring of such applications involves collecting values of metrics, e.g. performance related metrics, from a large number of member nodes to determine the state of the application or the system. We refer to such monitoring tasks as *application state monitoring*. Application state monitoring is essential for the observation, analysis and control of distributed applications and systems. For instance, data stream applications may require monitoring the data receiving/sending rate, captured events, tracked data entities, signature of internal states and any number of application-specific attributes on participating computing nodes to ensure stable operation in the face of highly bursty workloads [15][23]. Application provisioning may also require continuously collecting performance attribute values such as CPU usage, memory usage and packet size distributions from application-hosting servers [90].

One central problem in application state monitoring is organizing nodes into a certain topology where metric values from different nodes can be collected and delivered. In many cases, it is useful to collect detailed performance attributes at a controlled collection frequency. As an example, fine-grained performance characterization information is required to construct various system models and to test hypotheses on system behavior [53]. Similarly, the data rate and buffer occupancy in each element of a distributed application may be required for diagnosis purposes when there is a perceived bottleneck [15]. However, the overhead of collecting monitoring data grows quickly as the scale and complexity of monitoring tasks increase. Hence, it is crucial that the monitoring topology should ensure good monitoring scalability and cost-effectiveness at the same time.

While a set of monitoring-topology planning approaches have been proposed in the past, we find that these approaches often have the following drawbacks in general. First of all, existing works either build monitoring topologies for each individual monitoring task (TAG [71], SDIMS [123], PIER [50], join aggregations [33], REED [10], operator placement [102]), or use a static monitoring topology for all monitoring tasks [102]. These two approaches, however, often produce sub-optimal monitoring topologies. For example, if two monitoring tasks both collect metric values over the same set of nodes, using one monitoring tree for monitoring data transmission is more efficient than using two, as nodes can merge updates for both tasks and reduce per-message processing overhead. Hence, multi-monitoring-task level topology optimization is crucial for monitoring scalability.

Second, for many data-intensive environments, monitoring overhead grows substantially with the increase of monitoring tasks and deployment scale [123][87]. It is important that the monitoring topology should be resource sensitive, i.e. it should avoid monitoring nodes spending excessive resources on collecting and delivering attribute values. Unfortunately, existing works do not take node-level resource consumption as a first-class consideration. This may result in overload on certain nodes which eventually leads to monitoring data loss. Moreover, some assumptions in existing works do not hold in real world scenarios. For example, many works assume that the cost of update messages is only related with the number of values within the message, while we find that a fixed per-message overhead is not negligible.

Last but not the least, application state monitoring tasks are often subject to change in real world deployments [64]. Some tasks are short-term by nature, e.g. ad-hoc tasks submitted to check the current system usage [62]. Other tasks may be frequently modified for debugging, e.g. a user may specify different attributes for one task to understand which attribute provides the most useful information [64]. Nevertheless, existing works often consider monitoring tasks to be static and perform one-time topology optimization [10][102]. With little support for efficient topology adaptation, these approaches would either produce sub-optimal topologies when using a static topology regardless of changes in tasks, or introduce high adaptation cost when performing comprehensive topology reconstruction for any change in tasks [79].

In this chapter, we present REMO, a resource-aware application state monitoring system, that aims at addressing the above issues. REMO takes node-level available resources as the first class factor for building a monitoring topology. It optimizes the monitoring topology to achieve the best scalability and ensures that no node would be assigned with excessive monitoring workloads for their available resources.

REMO employs three key techniques to deliver cost-effective monitoring topologies under different environments. we first introduced a *basic topology planning algorithm*. This algorithm produces a forest of carefully optimized monitoring trees for a set of static monitoring tasks. It iteratively explores cost-sharing opportunities among monitoring tasks and refines the monitoring trees to achieve the best performance given the resource constraints on each node. One limitation of the basic approach is that it explores the entire search space for an optimal topology whenever the set of monitoring tasks is changed. This could lead to significant resource consumption for monitoring environments where tasks are subject to change. We then present an *adaptive topology planning algorithm* which continuously optimizes the monitoring topology according to the changes of tasks. To achieve cost-effectiveness, it maintains a balance between the topology adaptation cost and the topology efficiency, and employs cost-benefit throttling to avoid trivial adaptation. To ensure the efficiency and applicability of REMO, we also introduce a set of *optimization and extension techniques*. These techniques further improve the efficiency of resource-sensitive monitoring tree construction scheme, and allow REMO to support popular monitoring features such as in-network aggregation and reliability enhancements.

We undertake an experimental study of our system and present results including those gathered by deploying REMO on a BlueGene/P rack (using 256 nodes booted into Linux) running IBM's large-scale distributed streaming system - System S [15]. The results show that our resource-aware approach for application state monitoring consistently outperforms the current best known schemes. For instance, in our experiments with a real application that spanned up to 200 nodes and about as many monitoring tasks, using REMO to collect attributes resulted in a 35%-45% reduction in the percentage error of the attributes that were collected.

To our best knowledge, REMO is the first system that promotes resource-aware methodology to support and scale multiple application state monitoring tasks in large-scale distributed systems. We make three contributions in this chapter:

- We identify three critical requirements for large-scale application state monitoring: *the sharing of message processing cost among attributes*, *meeting node-level resource constraints*, and *efficient adaptation towards monitoring task changes*. Existing approaches do not address these requirements well.

- We propose a framework for communication-efficient application state monitoring. It allows us to optimize monitoring topologies to meet the above three requirements under a single framework.

- We develop techniques to further improve the applicability of REMO in terms of runtime efficiency and supporting new monitoring features.

Compared with recent works [118, 66] that study flexible architectures for tradeoff between monitoring/analysis costs and the benefits of monitoring/analysis results, we consider primarily CPU resource consumption related to monitoring communication or data collection and focus on developing concrete distributed monitoring algorithms that minimizes monitoring communication or data collection for a specific form of monitoring (state monitoring). In contrast, these works consider monitoring cost in terms of capital cost of dedicated monitoring hardware or software and aim at designing a flexible monitoring/analysis architecture. Furthermore, although our problem bears a superficial resemblance to distributed query optimization problems [63], our problem is fundamentally different since in our problem individual nodes are capacity constrained.

The rest of the chapter is organized as follows. Section 4.2 identifies challenges in application state monitoring. Section 6.3 illustrates the basic monitoring topology construction algorithm, and Section 2.4 introduces the adaptive topology construction algorithm. We optimize the efficiency of REMO and extend it for advanced features in Section 2.5 and 2.6. We present our experimental results in Section 4.7. Section 6.5 describes related works.

2.2 System Overview

In this section, we introduce the concept of application state monitoring and its system model. We also demonstrate the challenges in application state monitoring, and point out the key questions that an application state monitoring approach must address.

2.2.1 Application State Monitoring

Users and administrators of large-scale distributed applications often employ application state monitoring for observation, debugging, analysis and control purposes. Each application state monitoring task periodically collects values of certain attributes from the set of computing nodes over which an application is running. We use the term attribute and metric interchangeably in this chapter. As we focus on monitoring topology planning rather than the actual production of attribute values [74], we assume values of attributes are made available by application-specific tools or management services. In addition, we target at datacenter-like monitoring environments where any two nodes can communicate with similar cost (more details in Section 2.3.3). Formally, we define an application state monitoring task t as follows:

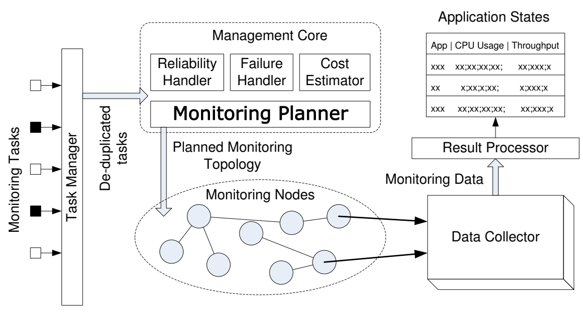


Figure 1: A High-Level System Model

Definition 1 *A monitoring task* $t = (A_t, N_t)$ *is a pair of sets, where* $A_t \subseteq \bigcup_{i \in N_t} A_i$ *is a set of attributes and* $N_t \subseteq N$ *is a set of nodes. In addition,* t *can also be represented as a list of node-attribute pairs* (i, j)*, where* $i \in N_t, j \in A_t$*.*

2.2.2 The Monitoring System Model

Figure 1 shows the high level model of REMO, a system we developed to provide application state monitoring functionality. REMO consists of several fundamental components:

Task manager takes state monitoring tasks and removes duplication among monitoring tasks. For instance, monitoring tasks $t_1 = (\{cpu\_utilization\}, \{a, b\})$ and $t_2 = (\{cpu\_utilization\}, \{b, c\})$ have duplicated monitored attribute $cpu\_utilization$ on node b. With such duplication, node b has to send $cpu\_utilization$ information twice for each update, which is clearly unnecessary. Therefore, given a set of monitoring tasks, the task manager transforms this set of tasks into a list of node-attribute pairs and eliminates all duplicated node-attribute pairs. For instance, t_1 and t_2 are equivalent to the list $\{a$-$cpu\_utilization$, b-$cpu\_utilization\}$ and $\{b$-$cpu\_utilization$, c-$cpu\_utilization\}$ respectively. In this case, node-attribute pair $\{b$-$cpu\_utilization\}$ is duplicated, and thus, is eliminated from the output of the task manager.

Management core takes de-duplicated tasks as input and schedules these tasks to run. One key sub-component of the management core is the *monitoring planner* which determines the inter-connection of monitoring nodes. For simplicity, we also refer to the overlay connecting monitoring nodes as the *monitoring topology*. In addition, the management core also provides important support for reliability enhancement and failure handling. **Data collector** provides a library of functions and algorithms for efficiently collecting attribute values from the monitoring network. It also serves as the repository of monitoring data and provides monitoring data access to users and high-level applications. **Result processor** executes the concrete monitoring operations including collecting and aggregating attribute values, triggering warnings, etc.

In this chapter, we focus on the design and implementation of the monitoring planner. We next introduce monitoring overhead in application state monitoring which drives the design principles of the monitoring planner.

2.2.3 Monitoring Overhead and Monitoring Planning

On a high level, a monitoring system consists of n monitoring nodes and one central node, i.e. data collector. Each monitoring node has a set of observable attributes $A_i = \{a_j | j \in [1, m]\}$. Attributes at different nodes but with the same subscription are considered as attributes of the same type. For instance, monitored nodes may all have locally observable CPU utilization. We consider an attribute as a continuously changing variable which outputs a new value in every unit time. For simplicity, we assume all attributes are of the same size a and it is straightforward to extend our work to support attributes with different sizes.

Each node i, the central node or a monitoring node, has a capacity b_i (also referred to as the resource constraint of node i) for receiving and transmitting monitoring data. In this chapter, we consider CPU as the primary resource for optimization. We associate each message transmitted in the system with a per-message overhead C, and, the cost of transmitting a message with x values is $C + ax$. This cost model is motivated by our observations of monitoring resource consumption on a real world system which we introduce next.

Our cost model considers both per-message overhead and the cost of payload. Although other models may consider only one of these two, our observation suggests that both costs should be captured in the model. Figure 2 shows how significant the per-message processing overhead is. The measurements were performed on a BlueGene/P node which has a 4-core 850MHz PowerPC processor. The figure shows an example monitoring task where nodes are configured in a star network where each node periodically transmits a single fixed small message to a root node over TCP/IP. The CPU utilization of the root node grows roughly linearly from around 6% for 16 nodes (the root receives 16 messages periodically) to around 68% for 256 nodes (the root receives 256 messages periodically). Note that this increased overhead is due to the increased number of messages at the root node and not due to the increase in the total size of messages. Furthermore, the cost incurred to receive a single message increases from 0.2% to 1.4% when we increase the number of values in the message from 1 to 256. Hence, we also model the cost associated with message size as a message may contain a large number of values relayed for different nodes.

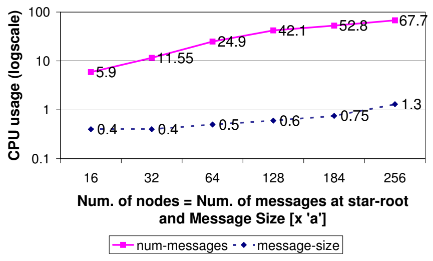


Figure 2: CPU Usage vs Increasing Message Number/Size

In other scenarios, the per-message overhead could be transmission or protocol overhead. For instance, a typical monitoring message delivered via TCP/IP protocol has a message header of at least 78 bytes not including application-specific headers, while an integer monitoring data is just 4 bytes.

As Figure 3 shows, given a list of node-attribute pairs, the monitoring planner organizes monitoring nodes into a forest of monitoring trees where each node collects values for a set of attributes. The planner considers the aforementioned per-message overhead as well as the cost of attributes transmission (as illustrated by the black and white bar in the left monitoring tree) to avoid overloading certain monitoring nodes in the generated monitoring

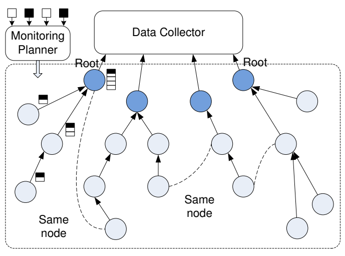


Figure 3: An Example of Monitoring Planning

topology. In addition, it also optimizes the monitoring topology to achieve maximum monitoring data delivery efficiency. As a result, one monitoring node may connect to multiple trees (as shown in Figure 3 and 4(c)). Within a monitoring tree T, each node i periodically sends an update message to its parent. As application state monitoring requires collecting values of certain attributes from a set of nodes, such update messages include both values locally observed by node i and values sent by i's children, for attributes monitored by T. Thus, the size of a message is proportional to the number of monitoring nodes in the subtree rooted at node i. This process continues upwards in the tree until the message reaches the central data collector node.

2.2.4 Challenges in Monitoring Planning

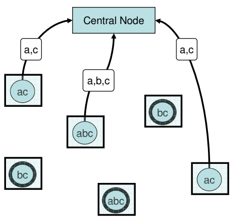
From the users' perspective, monitoring results should be as accurate as possible, suggesting that the underlying monitoring network should maximize the number of node-attribute pairs received at the central node. In addition, such a monitoring network should not cause the excessive use of resource at any node. Accordingly, we define the monitoring planning problem (MP) as follows:

Problem Statement 1 *Given a set of node-attribute pairs for monitoring* $\Omega = \{\omega_1, \omega_2, \ldots, \omega_p\}$ *where* $\omega_q = (i, j)$, $i \in N$, $j \in A$, $q \in [1, p]$, *and resource constraint* b_i *for each associated node, find a parent* $f(i, j), \forall i, j$, *where* $j \in A_i$ *such that node* i *forwards attribute* j *to node* $f(i, j)$ *that maximizes the total number of node-attribute pairs received at the central node and the resource demand of node* i, d_i, *satisfies* $d_i \leq b_i, \forall i \in N$.

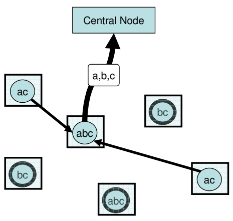
NP-completeness. When restricting all nodes to only monitor the same attribute j, we obtain a special case of the monitoring planning problem where each node has at most one attribute to monitor. As shown by Kashyap, et. al. [58], this special case is an NP-complete problem. Consequently, the monitoring planning problem (MP) is an NP-Complete problem, since each instance of MP can be restricted to this special case. Therefore, in REMO, we primarily focus on efficient approaches that can deliver reasonably good monitoring plan.

We now use some intuitive examples to illustrate the challenges and the key questions that need to be addressed in designing a resource-aware monitoring planner. Figure 4 shows a monitoring task involving 6 monitoring nodes where each node has a set of attributes to deliver (as indicated by alphabets on nodes). The four examples (a)(b)(c)(d) demonstrate different approaches to fulfill this monitoring task. Example (a) shows a widely used topology in which every node sends its updates directly to the central node. Unfortunately, this topology has poor scalability, because it requires the central node to have a large amount of resources to account for per-message overhead. We refer to the approach used in example (a) as the *star collection*. Example (b) organizes all monitoring nodes in a single tree which delivers updates for all attributes. While this monitoring plan reduces the resource consumption (per-message overhead) at the central node, the root node now has to relay updates for all node-attribute pairs, and again faces scalability issues due to limited resources. We refer to this approach as *one-set collection*. These two examples suggest that achieving certain degree of load balancing is important for a monitoring network.

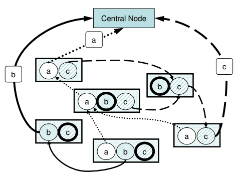
However, load balance alone does not lead to a good monitoring plan. In example (c), to balance the traffic among nodes, the central node uses three trees, each of which delivers only one attribute, and thus achieves a more balanced workload compared with example (b) (one-set collection) because updates are relayed by three root nodes. However, since each node monitors at least two attributes, nodes have to send out multiple update messages instead of one as in example (a) (star collection). Due to per-message overhead, this plan



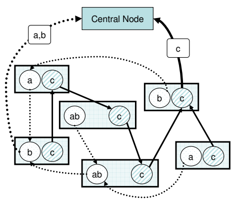
(a) Star collection



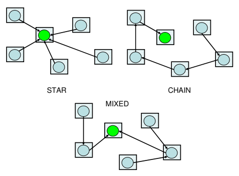
(b) One-Set collection



(c) Singleton-Set collection



(d) Optimal collection



(e) Different Trees

Figure 4: Motivating examples for the topology planning problem.

leads to higher resource consumption at almost *every* node. As a result, certain nodes may still fail to deliver all updates and less resources will be left over for additional monitoring tasks. We refer to the approach in example (c) as *singleton-set collection*.

The above examples reveal two fundamental aspects of the monitoring planning problem. First, *how to determine the number of monitoring trees and the set of attributes on each?* This is a non-trivial problem. Example (d) shows a topology which uses one tree to deliver attribute a,b and another tree to deliver attribute c. It introduces less per-message overhead compared with example (c) (singleton-set collection) and is a more load-balanced solution compared with example (b) (one-set collection). Second, *how to determine the topology for nodes in each monitoring tree under node level resource constraints?* Constructing monitoring trees subject to resource constraints at nodes is also a non-trivial problem and the choice of topology can significantly impact node resource usage. Example (e) shows three different trees. The star topology (upper left), while introducing the least relaying cost, causes significant per-message overhead at its root. The chain topology (upper right), on the contrary, distributes the per-message overhead among all nodes, but causes the most relaying cost. A "mixed" tree (bottom) might achieve a good trade-off between relaying cost and per-message overhead, but it is determine its optimal topology.

2.3 The Basic REMO Approach

The basic REMO approach promotes the resource aware multi-task optimization framework, consisting of a two phase iterative process and a suite of multi-task optimization techniques. At a high level, REMO operates as a *guided local search* approach, which starts with an initial monitoring network composed of multiple independently constructed monitoring trees, and iteratively optimizes the monitoring network until no further improvements are possible. When exploring various optimization directions, REMO employs cost estimation to guide subsequent improvement so that the search space can be restricted to a small size. This guiding feature is essential for the scalability of large-scale application state monitoring systems.

Concretely, during each iteration, REMO first runs a *partition augmentation* procedure which generates a list of most promising candidate augmentations for improving the current distribution of monitoring workload among monitoring trees. While the total number of candidate augmentations is very large, this procedure can trim down the size of the candidate list for evaluation by selecting the most promising ones through cost estimation. Given the generated candidate augmentation list, the *resource-aware evaluation* procedure further refines candidate augmentations by building monitoring trees accordingly with a resource-aware tree construction algorithm. We provide more details to these two procedures in the following discussion.

2.3.1 Partition Augmentation

The partition augmentation procedure is designed to produce the attribute partitions that can potentially reduce message processing cost through a guided iterative process. These attribute partitions determine the number of monitoring trees in the forest and the set of attributes each tree delivers. To better understand the design principles of our approach, we first briefly describe two simple but most popular schemes, which essentially represent the state-of-the-art in multiple application state monitoring.

Recall that among example schemes in Figure 4, one scheme (example (c)) delivers each attribute in a separate tree, and the other scheme (example (b)) uses a single tree to deliver updates for all attributes. We refer to these two schemes as the *Singleton-set partition scheme (SP)* and the *One-set partition (OP) scheme* respectively. We use the term "partition" because these schemes partition the set of monitored attributes into a number of non-overlapping subsets and assign each subsets to a monitoring tree.

Singleton-Set Partition (SP). Specifically, given a set of attributes for collection A, singleton-set partition scheme divides A into $|A|$ subsets, each of which contains a distinct attribute in A. Thus, if a node has m attributes to monitor, it is associated with m trees. This scheme is widely used in previous work, e.g. PIER [50], which constructs a routing tree for each attribute collection. While this scheme provides the most balanced load among trees, it is not efficient, as nodes have to send update messages for each individual attribute.

One-Set Partition (OP). The one-set partition scheme uses the set A as the only partitioned set. This scheme is also used in a number of previous work [102]. Using OP, each node can send just one message which includes all the attribute values, and thus, saves per-message overhead. Nevertheless, since the size of each message is much larger compared with messages associated with SP, the corresponding collecting tree can not grow very large, i.e. contains limited number of nodes.

2.3.1.1 Exploring Partition Augmentations

REMO seeks a middle ground between these extreme solutions - one where nodes pay lower per-message overhead compared to SP while being more load-balanced and consequently more scalable than OP. Our partition augmentation scheme explores possible augmentations to a given attribute partition P by searching for all partitions that are *close* to P in the sense that the resulting partition can be created by modifying P with certain predefined operations.

We define two basic operations that are used to modify attribute set partitions.

Definition 2 *Given two attribute sets A_i^P and A_j^P in partition P, a **merge** operation over A_i^P and A_j^P, denoted as $A_i^P \bowtie A_j^P$, yields a new set $A_k^P = A_i^P \cup A_j^P$. Given one attribute set A_i^P and an attribute α, a **split** operation on A_i^P with regard to α, denoted as $A_i^P \triangleright \alpha$, yields two new sets $A_k^P = A_i^P - \alpha$.*

A merge operation is simply the union of two set attributes. A split operation essentially removes one attribute from an existing attribute set. As it is a special case of set difference operation, we use the set difference sign $(-)$ here to define split. Furthermore, there is no restriction on the number of attributes that can be involved in a merge or a split operation. Based on the definition of merge and split operations, we now define neighboring solution as follows:

Definition 3 *For an attribute set partition P, we say partition P' is a* ***neighboring solution*** *of P if and only if either $\exists A_i^P, A_j^P \in P$ so that $P' = P - A_i^P - A_j^P + (A_i^P \bowtie A_j^P)$, or $\exists A_i^P \in P, \alpha \in A_i^P$ so that $P' = P - A_i^P + (A_i^P \rhd \alpha) + \{\alpha\}$.*

A neighboring solution is essentially a partition obtained by make “one-step” modification (either one merge or one split operation) to the existing partition.

Guided Partition Augmentation. Exploring all neighboring augmentations of a given partition and evaluating the performance of each augmentation is practically infeasible, since the evaluation involves constructing resource-constrained monitoring trees. To mitigate this problem, we use a guided partition augmentation scheme which greatly reduces the number of candidate partitions for evaluation. The basic idea of this guided scheme is to rank candidate partitions according to the *estimated* reduction in the total capacity usage that would result from using the new partition. The rationale is that a partition that provides a large decrease in capacity usage will free up capacity for more attribute value pairs to be aggregated. Following this, we evaluate neighboring partitions in the decreased order of their estimated capacity-reduction so that we can find a good augmentation without evaluating all candidates.

To estimate the gain of a candidate augmentation, we first need to understand how the resource consumption would change after applying an augmentation m. Change in the total resource consumption resulting from an augmentation m can be contributed by the change in the relay cost and that in the per-message overhead cost, as m may change the number of trees and the structure of trees. Therefore, let $g(m)$ be the overall reduction in resource consumption of an augmentation m, $\Delta c_p(m)$ be the estimated difference in overhead cost due to m and $\Delta c_r(m)$ be the estimated difference in relay cost due to m. We then have $g(m) = \Delta c_p(m) + \Delta c_r(m)$. We estimate $g(m)$ assuming that following an augmentation, the tree construction procedure is able to assign *all* the attribute-value pairs that were in the affected partitions using a *star* topology. Assuming a topology is necessary to be able to estimate $\Delta c_r(m)$. Recall that C is the per-message overhead and a is the cost of a message of unit size. Also let N_{A_i} denote the number of nodes associated with attribute set A_i. We then have:

$$
\begin{aligned}
\Delta c_p(m) &= \begin{cases} (-1) \cdot C \cdot |N_{A_i} \cap N_{A_j}| & m : A_i \bowtie A_j = A_k \\ C \cdot |N_{A_j} \cap N_{A_k}| & m : A_i \rhd A_j = A_k \end{cases} \\
\Delta c_r(m) &= \begin{cases} a \cdot |N_{A_i \cup A_j} - N_{A_i \cap A_j}| & m : A_i \bowtie A_j = A_k \\ (-1) \cdot a \cdot |N_{A_i} - N_{A_i \cap A_j}| & m : A_i \rhd A_j = A_k \end{cases}
\end{aligned}
$$

Intuitively, when we merge two attribute sets, the per-message overhead cost reduces as nodes associated with both sets need to send fewer messages for an update. However, the corresponding relaying cost may increase since the merged tree may be higher than the previous two trees, which in turn makes messages travel more hops to reach the root node. On the contrary, when we split an attribute set, the per-message overhead cost increases and the relaying cost decreases. The above equations capture these two changes and make the estimation possible. This guided local-search heuristic is essential to ensuring the practicality of our scheme.

2.3.2 Resource-aware Evaluation

To evaluate the objective function for a given candidate partition augmentation m, the resource-aware evaluation procedure evaluates m by constructing trees for nodes affected by m and measures the the total number of node-attribute value pairs that can be collected using these trees. This procedure primarily involves two tasks. One is *constructing a tree* for a given set of nodes without exceeding resource constraints at any node. The other is for a node connected to multiple trees to *allocate its resources* to different trees.

2.3.2.1 Tree Construction

The tree construction procedure constructs a collection tree for a given set of nodes D such that no node exceeds its resource constraints while trying to include as many nodes as possible into the constructed tree. Formally, we define the tree construction problem as follows:

Problem Statement 2 *Given a set of n vertices, each has x_i attributes to monitor, and resource constraint b_i, find a parent vertex $p(i), \forall i$, so that the number of vertices in the constructed tree is maximized subject to the following constraints where u_i is the resource consumed at vertex i for sending update messages to its parent:*

1. *For any vertex i in the tree, $\sum_{p(j)=i} u_j + u_i \leq b_i$*
2. *Let y_i be the number of all attribute values transmitted by vertex i. We have $y_i = x_i + \sum_{p(j)=i} x_j$.*
3. *According to our definition, $u_i \leq C + y_i \cdot a$*

The first constraint requires that the resource spent on node i for sending and receiving updates should not exceed its resource constraint b_i. The second constraint requires a node to deliver its local monitored values as well as values received from its children. The last constraint states that the cost of processing an outgoing message is the combination of per-message overhead and value processing cost. The tree construction problem, however, is also NP-Complete [58] and we present heuristics for the tree-construction problem.

To start with, we first discuss two simple tree construction heuristics:

Star. This scheme forms "star"-like trees by giving priority to increasing the breadth of the tree. Specifically, it adds nodes into the constructed tree in the order of decreased available capacity, and attaches a new node to the node with *the lowest height* and sufficient available capacity, until no such nodes exist. STAR creates bushy trees and consequently pays low relay cost. However, owing to large node degrees, the root node suffers from higher per-message overhead, and consequently, the tree can not grow very large.

Chain. This scheme gives priority to increasing the height of the tree, and constructs “chain”-like trees. CHAIN adds nodes to the tree in the same way as STAR does except that it tries to attach nodes to the node with *the highest height* and sufficient available capacity. CHAIN creates long trees that achieve very good load balance, but due to the number of hops each message has to travel to reach the root, most nodes pay a high relay cost.

STAR and CHAIN reveal two conflicting factors in collection tree construction – resource efficiency and scalability. Minimizing tree height achieves resource efficiency, i.e. minimum relay cost, but causes poor scalability, i.e. small tree size. On the other hand, maximizing tree height achieves good scalability, but degrades resource efficiency. The adaptive tree construction algorithm seeks a middle-ground between the STAR and CHAIN procedures in this regard. It tries to minimize the total resource consumption, and can trade off overhead cost for relay cost, and vice versa, if it is possible to accommodate more nodes by doing so.

Before we describe the adaptive tree construction algorithm, we first introduce the concept of saturated trees and congested nodes as follows:

Definition 4 *Given a set of nodes N for tree construction and the corresponding tree T which contains a set of nodes $N' \subset N$, we say T is* ***saturated*** *if no more nodes $d \in (N-N')$ can be added to T without causing the resource constraint to be violated for at least one node in T. We refer to nodes whose resource constraint would be violated if $d \in (N-N')$ is added to T as* ***congested*** *nodes.*

The adaptive tree construction algorithm iteratively invokes two procedures, the *construction* procedure and the *adjusting* procedure. The construction procedure runs the STAR scheme which attaches new nodes to low level existing tree nodes. As we mentioned earlier, STAR causes the capacity consumption at low level tree nodes to be much heavier than that at other nodes. Thus, as low level tree nodes become congested we get a saturated tree, the construction procedure terminates and returns all congested nodes. The algorithm then invokes the adjusting procedure, which tries to relieve the workload of low level tree nodes by reducing the degree of these nodes and increasing the height of the tree(similar to CHAIN). As a result, the adjusting procedure reduces congested nodes and makes a saturated tree unsaturated. The algorithm then repeats the constructing-adjusting iteration until no more nodes can be added to the tree or all nodes have been added.

2.3.3 Discussion

REMO targets at datacenter-like environments where the underlying infrastructure allows any two nodes in the network can communicate with similar cost, and focuses on the resource consumption on computing nodes rather than that of the underlying network. We believe this setting fits for many distributed computing environments, even when computing nodes are not directly connected. For instance, communication packets between hosts located in the same rack usually pass through only one top-of-rack switch, while communication packets between hosts located in different racks may travel through longer communication path consisting of multiple switches or routers. The corresponding overhead on communication endpoints, however, is similar in these two cases as packet forwarding overhead is outsourced to network devices. As long as networks are not saturated, REMO can be directly applied for monitoring topology planning.

when the resource consumption on network devices needs to be considered, e.g. networks are bottleneck resources, REMO cannot be directly applied. Similarly, for environments where inter-node communication requires nodes to actively forward messages, e.g. peer-to-peer overlay networks and wireless sensor networks, the assumption of similar cost on communication endpoints does not hold as longer communication paths also incur higher forwarding cost. However, REMO can be extended to handle such changes. For example, its local search process can incorporate the forwarding cost in the resource evaluation of a candidate plan. We consider such extension for supporting such networks as our future work.

2.4 Runtime Topology Adaption

The basic REMO approach works well for a static set of monitoring tasks. However, in many distributed computing environments, monitoring tasks are often added, modified or removed on the fly for better information collection or debugging. Such changes necessitate the adaptation of monitoring topology. In this section, we study the problem of runtime topology adaptation for changes in monitoring tasks.

2.4.1 Efficient Adaptation Planning

One may search for an optimal topology by invoking the REMO planning algorithm every time a monitoring task is added, modified or removed, and update the monitoring topology accordingly. We refer to such an approach as *REBUILD*. REBUILD, however, may incur significant resource consumption due to topology planning computation as well as topology reconstruction cost (e.g. messages used for notifying nodes to disconnect or connect), especially in datacenter-like environments with a massive number of mutable monitoring tasks undergoing relatively frequent modification.

An alternative approach is to introduce minimum changes to the topology to fulfill the changes of monitoring tasks. We refer to such an approach as *DIRECT-APPLY* or *D-A* for brevity. D-A also has its limitation as it may result in topologies with poor performance over time. For instance, when we continuously add attributes to a task for collection, D-A simply instructs the corresponding tree to deliver these newly added attribute values until some nodes become saturated due to increased relay cost.

To address such issues, we propose an efficient adaptive approach that strikes a balance between adaptation cost and topology performance. The basic idea is to look for new topologies with good performance and small adaptation cost (including both searching and adaptation-related communication cost) based on the modification to monitoring tasks. Our approach limits the search space to topologies that are close variants of the current topology in order to achieve efficient adaptation. In addition, it ranks candidate adaptation operations based on their estimated cost-benefit ratios so that it always performs the most worthwhile adaptation operation first. We refer to this scheme as *ADAPTIVE* for brevity.

When monitoring tasks are added, removed or modified, we first applies D-A by building the corresponding trees with the tree building algorithm introduced in Section 6.3 (no changes in the attribute partition). We consider the resulting monitoring topology after invoking D-A as *the base topology*, which is then optimized by our ADAPTIVE scheme. Note that the base topology is only a virtual topology plan stored in memory. The actual monitoring topology is updated only when the ADAPTIVE scheme produces a final topology plan.

Same as the algorithm in Section 6.3, the ADAPTIVE scheme performs two operations, merging and splitting, over the base topology in an iterative manner. For each iteration, the ADAPTIVE scheme first lists all candidate adaptation operations for merging and splitting respectively, and ranks the candidate operations based on estimated cost-effectiveness. It then evaluates merging operations in the order of decreasing cost-effectiveness until it finds a valid merging operation. It also evaluates splitting operations in the same way until it finds a valid splitting operations. From these two operations, it chooses one with the largest improvement to apply to the base topology.

Let T be the set of reconstructed trees. To ensure efficiency, the ADAPTIVE scheme considers only merging operations involving at least one tree in T as candidate merging operations. This is because merging two trees that are not in T is unlikely to improve the topology. Otherwise, previous topology optimization process would have adopted such a merging operation. The evaluation of a merging operation involves computationally expensive tree building. As a result, evaluating only merging operations involving trees in T greatly reduces the search space and ensures the efficiency and responsiveness (changes of monitoring tasks should be quickly applied) of the ADAPTIVE scheme. For a monitoring topology with n trees, the number of possible merging operations is $C_n^2 = \frac{n(n-1)}{2}$, while the number of merging operations involving trees in T is $|T| \cdot (n-1)$ which is usually significantly smaller than $\frac{n(n-1)}{2}$ as $T << n$ for most monitoring task updates. Similarly, the ADAPTIVE scheme considers a splitting operation as a candidate operation only if the tree to be split is in T.

The ADAPTIVE scheme also minimizes the number of candidate merging/splitting operations it evaluates for responsiveness. It ranks all candidate operations and always evaluates the one with the greatest potential gain first. To rank candidate operations, the ADAPTIVE scheme needs to estimate the cost-effectiveness of each operation. We estimate the cost-effectiveness of an operation based on its estimated benefit and estimated adaptation cost. The estimated benefit is the same as $g(m)$ we introduced in Section 6.3. The estimated adaptation cost refers to the cost of applying the merging operation to the existing monitoring topology. This cost is usually proportional to the the number of edges modified in the topology. To estimate this cost, we use the lower bound of the number of edges that would have to be changed.

2.4.2 Cost-Benefit Throttling

The ADAPTIVE scheme must ensure that the benefit of adaption justifies the corresponding cost. For example, a monitoring topology undergoing frequent modification of monitoring tasks may not be suitable for frequent topology adaptation unless the corresponding gain is substantial. We employ cost-benefit throttling to apply only topology adaptations whose gain exceeds the corresponding cost. Concretely, when the ADAPTIVE scheme finds a valid merging or splitting operation, it estimates the adaptation cost by measuring the volume of control messages needed for adaptation, denoted by M_{adapt}. The algorithm considers the operation cost-effective if M_{adapt} is less than a threshold defined as follows,

$$Threshold(A_m) = (T_{cur} - \min\{T_{adj,i}, i \in A_m\}) \cdot (C_{cur} - C_{adj})$$

, where A_m is the set of trees involved in the operation, $T_{adj,i}$ is the last time tree i being adjusted, T_{cur} is the current time, C_{cur} is the volume of monitoring messages delivered in unit time in the trees of the current topology, and C_{adj} is the volume of monitoring messages delivered in unit time in the trees after adaptation. $(T_{cur} - \min\{T_{adj,i}, i \in A_m\})$ essentially captures how frequently the corresponding trees are adjusted, and $(C_{cur} - C_{adj})$ measures the efficiency gain of the adjustment. Note that the threshold will be large if either the potential gain is large, i.e. $(C_{cur} - C_{adj})$ is large, or the corresponding trees are unlikely to be adjusted due to monitoring task updates, i.e. $(T_{cur} - \min\{T_{adj,i}, i \in A_m\})$ is large. Cost-benefit throttling also reduces the number of iterations. Once the algorithm finds that a merging or splitting is not cost-effective, it can terminate immediately.

2.5 *Optimization*

The basic REMO approach can be further optimized to achieve better efficiency and performance. In this section, we present two techniques, efficient tree adjustment and ordered resource allocation, to improve the efficiency of REMO tree construction algorithm and its planning performance respectively.

2.5.1 Efficient Tree Adjustment

The tree building algorithm introduced in Section 6.3 iteratively invokes a construction procedure and an adjusting procedure to build a monitoring tree for a set of nodes. One issue of this tree building algorithm is that it generates high computation cost, especially its adjusting procedure. To increase the available resource of a congested node d_c, the adjusting procedure tries to reduce its resource consumption on per-message overhead by reducing the number of its branches. Specifically, the procedure first removes the branch of d_c with the least resource consumption. We use b_{d_c} to denote this branch. It then tries to reattach nodes in b_{d_c} to other nodes in the tree except d_c. It considers the reattaching successful if all nodes of b_{d_c} is attached to the tree. As a result, the complexity of the adjusting procedure is $O(n^2)$ where n is the number of nodes in the tree.

We next present two techniques that reduces the complexity of the adjusting procedure. The first one, branch based reattaching, reduces the complexity to $O(n)$ by reattaching the entire branch b_{d_c} instead of individual nodes in b_{d_c}. It trades off a small chance of failing to reattaching b_{d_c} for considerable efficiency improvement. The second technique, Subtree-only searching, reduces the reattaching scope to the subtree of d_c, which considerably reduces searching time in practice (the complexity is still $O(n)$). The subtree-only searching also theoretically guarantees the searching completeness.

2.5.1.1 Branch Based Reattaching

The above adjusting procedure breaks branches into nodes and moves one node at a time. This per-node-reattaching scheme is quite expensive. To reduce the time complexity, we adopts a branch-based reattaching scheme. As its name suggests, this scheme removes a branch from the congested node and attaches it entirely to another node, instead of breaking the branch into nodes and performing the reattaching. Performing reattaching in a branch basis effectively reduces the complexity of the adjusting procedure.

One minor drawback of branch-based reattaching is that it diminishes the chance of finding a parent to reattach the branch when the branch consists of many nodes. However, the impact of this drawback is quite limited in practice. As the adjusting procedure removes and reattaches the smallest branch first, failing to reattaching the branch suggests that nodes of the tree all have limited available resource. In this case, node based reattaching is also likely to fail.

2.5.1.2 Subtree-Only Searching

The tree adjusting procedure tries reattaching the pruned branch to all nodes in the tree except the congested node denoted as d_c. This adjustment scheme is not efficient as it enumerates almost every nodes in the tree to test if the pruned branch can be reattached to the node. It turns out that testing nodes outside d_c's subtree is often unnecessary. The following theorem suggests that testing nodes within d_c's subtree is sufficient as long as the resource demand of the node failed to add is higher than that of the pruned branch.

Theorem 1 *Given a saturated tree T outputted by the construction procedure, d_f the node failed to add to T, a congested node d_c and one of its branches b_{d_c}, attaching b_{d_c} to any node outside the subtree of d_c causes overload, given that the resource demand of d_f is no larger than that of b_{d_c}, i.e. $u_{d_f} \leqslant u_{b_{d_c}}$*

Proof If there exists a node outside the subtree of d_c, namely d_o, that can be attached with b_{d_c} without causing overload, then adding d_f to d_o should have succeeded in the previous execution of the construction procedure, as $u_{d_f} \leqslant u_{b_{d_c}}$. However, T is a saturated tree when adding d_f, which leads to a contradiction. □

Hence, we improve the efficiency of the original tree building algorithm by testing all nodes for reattaching only when the resource demand of the node failed to add is higher than that of the pruned branch. For all other cases, the algorithm performs reattaching test only within the subtree of d_c.

2.5.2 Ordered Resource Allocation

For a node associated with multiple trees, determining how much resource it should assign to each of its associated trees is necessary. Unfortunately, finding the optimal resource allocation is difficult because it is not clear how much resource a node will consume until the tree it connects to is built. Exploring all possible allocations to find the optimal one is clearly not an option as the computation cost is intractable.

To address this issue, REMO employs an efficient on-demand allocation scheme. Since REMO builds the monitoring topology on a tree-by-tree sequential basis, the on-demand allocation scheme defers the allocation decision until necessary and only allocates capacity to the tree that is about to be constructed. Given a node, the on-demand allocation scheme assigns all current available capacity to the tree that is currently under construction. Specifically, given a node i with resource b_i and a list of trees each with resource demand d_{ij}, the available capacity assigned to tree j is $b_i - \sum_{k=1}^{j-1} d_{ij}$. Our experiment results suggest that our on-demand allocation scheme outperforms several other heuristics.

The on-demand allocation scheme has one drawback that may limit its performance when building trees with very different sizes. As on-demand allocation encourages the trees constructed first to consume as much resource as necessary, the construction of these trees may not invoke the adjusting procedure which saves resource consumption on parent nodes by reducing their branches. Consequently, resources left for constructing the rest of the trees is limited.

We employ a slightly modified on-demand allocation scheme that relieves this issue with little additional planning cost. Instead of not specifying the order of construction, the new scheme constructs trees in the order of increasing tree size. The idea behind this modification is that small trees are more cost-efficient in the sense that they are less likely to consume much resource for relaying cost. By constructing trees from small ones to large ones, the construction algorithm pays more relaying cost for better scalability only after small trees are constructed. Our experiment results suggest the ordered scheme outperforms the on-demand scheme in various settings.

2.6 Extensions

Our description of REMO so far is based on a simple monitoring scenario where tasks collect distributed values without aggregation or replication under a uniform value updating frequency. Real world monitoring, however, often poses diverse requirements. In this section, we present three important techniques to support such requirements in REMO. The in-network-aggregation-aware planning technique allows REMO to accurately estimate per-node resource consumption when monitoring data can be aggregated before being passed to parent nodes. The reliability enhancement technique provides additional protection to monitoring data delivery by producing topologies that replicate monitoring data and pass them through distinct paths. The heterogeneous-update-frequency supporting technique enables REMO to plan topologies for monitoring tasks with different data updating frequencies by correctly estimating per-node resource consumption of such mixed workloads. These three techniques introduce little to no extra planning cost. Moreover, they can be incorporated into REMO as plugins when certain functionality is required by the monitoring environment without modifying the REMO framework.

2.6.1 Supporting In-Network Aggregation

In-network aggregation is important to achieve efficient distributed monitoring. Compared with holistic collection, i.e. collecting individual values from nodes, In-network aggregation allows individual monitoring values to be combined into aggregate values during delivery. For example, if a monitoring task requests the SUM of certain metric m on a set of nodes N, with in-network aggregation, a node can locally aggregate values it receives from other nodes into a single partial sum and pass it to its parent, instead of passing each individual value it receives.

REMO can be extended to build monitoring topology for monitoring tasks with in-network aggregation. We first introduce a *funnel function* to capture the changes of resource consumption caused by in-network aggregation. Specifically, a funnel function on node i, $fnl_i^m(g_m, n_m)$, returns the number of outgoing values on node i for metric m given the in-network aggregation type g_m and the number of incoming values n_m. The corresponding resource consumption of node i in tree k for sending update message to its parent is,

$$u_{ik} = C + \sum_{m \in A_i \cap A_k^P} a \cdot fnl_i^m(g_m, n_m) \tag{1}$$

where $A_i \cap A_k^P$ is the set of metrics node i needs to collect and report in tree k, a is the per value overhead and C is the per message overhead. For SUM aggregation, the corresponding funnel function is $fnl_i^m(SUM_m, n_m) = 1$ because the result of SUM aggregation is always a single value. Similarly, for TOP10 aggregation, the funnel function is $fnl_i^m(TOP10_m, n_m) = \min\{10, n_m\}$. For holistic aggregation we discussed earlier,

$fnl_i^m(HOLISTIC_m, n_m) = n_m$. Hence, Equation 1 can be used to calculate per-node resource consumption for both holistic aggregation and in-network aggregation in the aforementioned monitoring tree building algorithm. Note that it also supports the situation where one tree performs both in-network and holistic aggregation for different metrics.

Some aggregation functions such as DISTINCT, however, are data dependent in terms of the result size. For example, applying DISTINCT on a set X of 10 values results in a set with size ranging from 1 to 10, depending how many repeated values A contains. For these aggregations, we simply employ the funnel function of holistic aggregation for an upper bound estimation in the current implementation of REMO. Accurate estimation may require sampling based techniques which we leave as our future work.

2.6.2 Reliability Enhancements

Enhancing reliability is important for certain mission critical monitoring tasks. REMO supports two modes of reliability enhancement, *same source different paths(SSDP)* and *different sources different paths(DSDP)*, to minimize the impact of link and node failure. The most distinct feature of the reliability enhancement in REMO is that the enhancement is done by rewriting monitoring tasks and requires little modification to the original approach.

The SSDP mode deals with link failures by duplicating the transmission of monitored values in different monitoring trees. Specifically, for a monitoring task $t = (a, N_t)$ requiring SSDP support, REMO creates a monitoring task $t' = (a', N_t)$ where a' is an alias of a. In addition, REMO restricts that a and a' would never occur in the same set of a partition P during partition augmentation, which makes sure messages updating a and a' are transmitted within different monitoring trees, i.e. different paths. Note that the degree of reliability can be adjusted through different numbers of duplications.

When a certain metric value is observable at multiple nodes, REMO also supports the DSDP mode. For example, computing nodes sharing the same storage observe the same storage performance metric values. Under this mode, users submit monitoring tasks in the form of $t = (a, N_{identical})$ where $N_{identical} = N(v_1), N(v_2), \ldots, N(v_n)$. $N(v_i)$ denotes the set of nodes that observe the same value v_i and $N_{identical}$ is a set of node groups each of which observes the same value. Let $k = \min\{|N(v_i)|, i \in [1, n]\}$. REMO rewrites t into k monitoring tasks so that each task collects values for metric a with a distinct set of nodes drawn from $N(v_i), i \in [1, n]$. Similar to SSDP model, REMO then constructs the topology by avoiding any of the k monitoring tasks to be clustered into one tree. In this way, REMO ensures values of metrics can be collected from distinct sets of nodes and delivered through different paths.

2.6.3 Supporting Heterogeneous Update Frequencies

Monitoring tasks may collect values from nodes with different update frequencies. REMO supports heterogeneous update frequencies by grouping nodes based on their metric collecting frequencies and constructing per-group monitoring topologies. When a node has a single metric a with the highest update frequency, REMO considers the node as having only one metric to update as other metrics piggyback on a. When a node has a set of metrics updated at the same highest frequencies, denoted by A_h, it evenly assigns other metrics to piggyback on metrics in of A_h. Similarly, REMO considers the node as having a set of metrics A_h to monitor as other metrics piggyback on A_h. We estimate the cost of updating with piggybacked metrics for node i by $u_i = C + a \cdot \sum_j freq_j / freq_{max}$ where $freq_j$ is the frequency of one metric collected on node i and $freq_{max}$ is the highest update frequency on node i.

Sometimes metric piggybacking cannot achieve the precise update frequency defined by users. For example, if the highest update frequency on a node is 1/5 (msg/sec), a metric updated at 1/22 can at best be monitored at either 1/20 or 1/25. If users are not satisfied with such an approximation, our current implementation separates these metrics out and builds individual monitoring trees for each of them.

2.7 Experimental Evaluation

We undertake an experimental study of our system and present results including those gathered by deploying REMO on a BlueGene/P rack (using 256 nodes booted into Linux) running IBM's large-scale distributed streaming system - System S.

Synthetic Dataset Experiments. For our experiments on synthetic data, we assign a random subset of attributes to each node in the system. For monitoring tasks, we generate them by randomly selecting $|A_t|$ attributes and $|N_t|$ nodes with uniform distribution, for a given size of attribute set A and node set N. We also classify monitoring tasks into two categories - 1) **small-scale** monitoring tasks that are for a small set of attributes from a small set of nodes, and 2) **large-scale** monitoring tasks that either involves many nodes or many attributes.

To evaluate the effectiveness of different topology construction schemes, we measure the percentage of attribute values collected at the root node with the monitoring topology produced by a scheme. Note that this value should be 100% when the monitoring workload is trivial or each monitoring node is provisioned with abundant monitoring resources. For comparison purposes, we apply relatively heavy monitoring workloads to keep this value below 100% for all schemes. This allows us to easily compare the performance of different schemes by looking at their percentage of collected values. Schemes with higher percentage of collected values not only achieve better monitoring coverage when monitoring resources are limited, but also have better monitoring efficiency in terms of monitoring resource consumption.

Real System Experiments. Through experiments in a real system deployment, we also show that the error in attribute value observations (due to either stale or dropped attribute values) introduced by REMO is small. Note that this error can be measured in a meaningful way only for a real system and is what any "user" of the monitoring system would perceive when using REMO.

System S is a large-scale distributed stream processing middleware. Applications are expressed as dataflow graphs that specify analytic operators interconnected by data streams. These applications are deployed in the System S runtime as processes executing on a distributed set of hosts, and interconnected by stream connections using transports such as TCP/IP. Each node that runs application processes can observe attributes at various levels such as at the analytic operator level, System S middleware level, and the OS level. For these experiments, we deployed one such System S application called *YieldMonitor* [107], that monitors a chip manufacturing test process and uses statistical stream processing to predict the yield of individual chips across different electrical tests. This application consisted of over 200 processes deployed across 200 nodes, with 30-50 attributes to be monitored on each node, on a BlueGene/P cluster. The BlueGene is very communication rich and all compute nodes are interconnected by a 3D Torus mesh network. Consequently, for all practical purposes, we have a fully connected network where all pairs of nodes can communicate with each other at almost equal cost.

2.7.1 Result Analysis

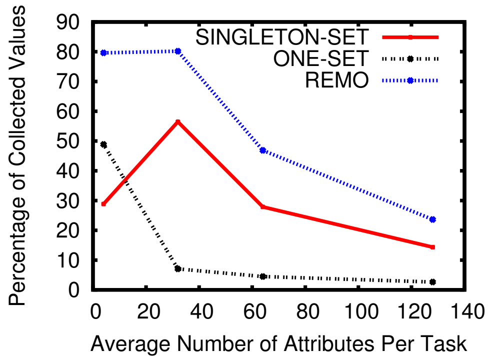
We present a small subset of our experimental results to highlight the following observations amongst others. First, REMO can collect a larger fraction of node-attribute pairs to serve monitoring tasks presented to the system compared to simple heuristics (which are essentially the state-of-the-art). REMO *adapts* to the task characteristics and outperforms each of these simple heuristics for all types of tasks and system characteristics, e.g. for small-scale tasks, a collection mechanism with fewer trees is better while for large-scale tasks, a collection mechanism with more trees is better. Second, in a real application scenario, REMO also significantly reduces percentage error in the observed values of the node-attribute pairs required by monitoring tasks when compared to simple heuristics.

Varying the scale of monitoring tasks. Figure 5 compares the performance of different attribute set partition schemes under different workload characteristics. In Figure 5(a), where we increase the number of attributes in monitoring tasks, i.e. increasing $|A_t|$, our partition augmentation scheme(REMO) performs consistently better than singleton-set(SINGLETON-SET) and one-set(ONE-SET) schemes. In addition, ONE-SET outperforms SINGLETON-SET when $|A_t|$ is relatively small. As each node only sends out one message which includes all its own attributes and those received from its children, ONE-SET causes the minimum per-message overhead. Thus, when each node monitors relatively small number of attributes, it can efficiently deliver attributes without suffering from its scalability problem. However, when $|A_t|$ increases, the capacity demand of the low level nodes, i.e. nodes that are close to the root, increases significantly, which in turn limits the size of the tree and causes poor performance. In Figure 5(b), where we set $|A_t| = 100$ and increase $|N_t|$ to create extremely heavy workloads, REMO gradually converges to SINGLETON-SET, as SINGLETON-SET achieves the best load balance under heavy workload which in turn results in the best performance.

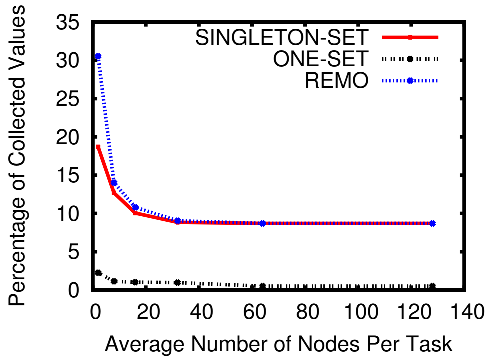
Varying the number of monitoring tasks. We observe similar results in Figure 5(c) and 5(d), where we increase the total number of small-scale and large-scale monitoring tasks respectively.

Varying nodes in the system. Figure 6 illustrates the performance of different attribute set partition schemes with changing system characteristics. In Figure 6(a) and 6(b), where we increase the number of nodes in the system given small and large scale monitoring tasks respectively, we can see SINGLETON-SET is better for large-scale tasks while ONE-SET is better for small-scale tasks, and REMO performs much better than them in both cases, around 90% extra collected node-attribute pairs.

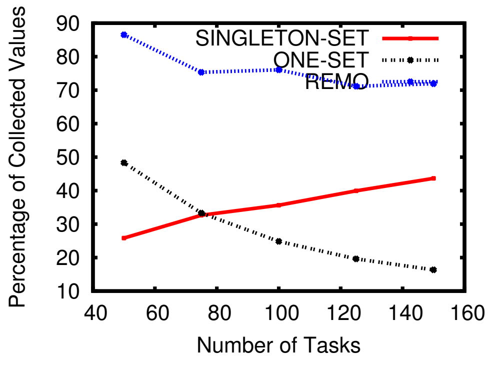
Varying per-message processing overhead. To study the impact of per-message overhead, we vary the C/a ratio under both small and large-scale monitoring tasks in Figure 6(c) and 6(d). As expected, increased per-message overhead hits the SINGLETON-SET scheme hard since it constructs a large number of trees and, consequently, incurs the largest overhead cost while the performance of the ONE-SET scheme which constructs just a single tree degrades more gracefully. However having a single tree is not the best solution as



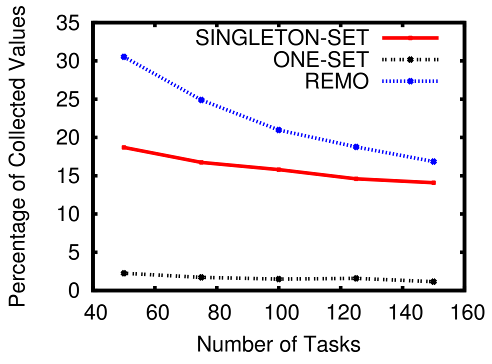
(a) Increasing $|A_t|$



(b) Increasing $|N_t|$

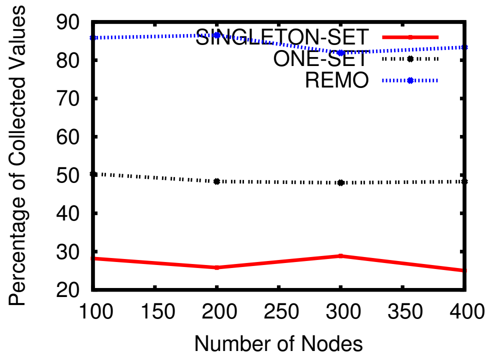


(c) Increasing Small-scale Tasks

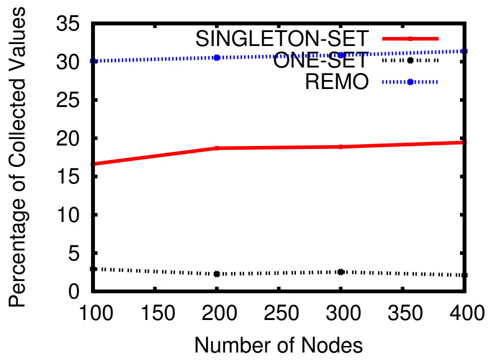


(d) Increasing Large-scale Tasks

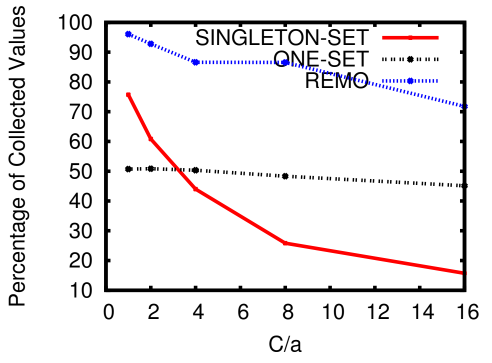
Figure 5: Comparison of Attribute Set Partition Schemes under Different Workload Characteristics



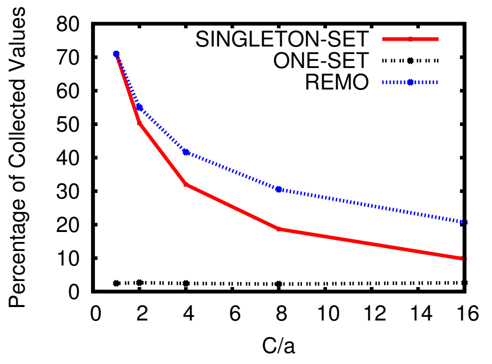
(a) Increasing Nodes(Small-scale)



(b) Increasing Nodes(Large-scale)



(c) Increasing C/a(Small-scale)



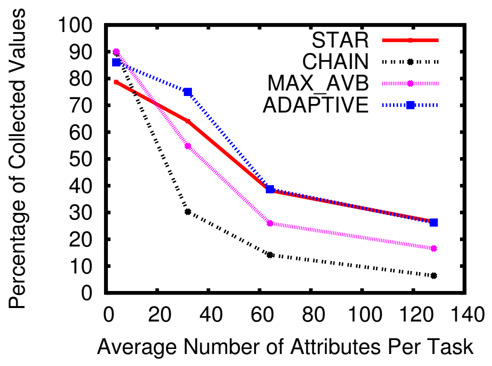
(d) Increasing C/a(Large-scale)

Figure 6: Comparison of Attribute Set Partition Schemes under Different System Characteristics

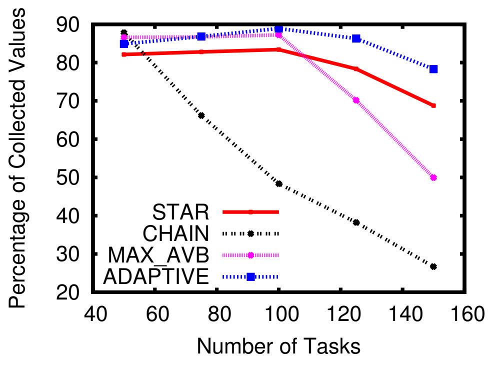
shown by REMO which outperforms both the schemes as C/a is increased, because it can reduce the number of trees formed when C/a is increased.

Comparison of tree-construction schemes. In Figure 7, we study the performance of different tree construction algorithms under different workloads and system characteristics. Our comparison also includes a new algorithm, namely *MAX_AVB*, a heuristic algorithm used in TMON [58] which always attaches new node to the existing node with the most available capacity. While we vary different workloads and system characteristics in the four figures, our adaptive tree construction algorithm(ADAPTIVE) always performs the best in terms of percentage of collected values. Among all the other tree construction schemes, STAR performs well when workload is heavy, as suggested by Figure 7(a) and 7(b). This is because STAR builds trees with minimum height, and thus, pays minimum cost for relaying, which can be considerable when workloads are heavy. CHAIN performs the worst in almost all cases. While CHAIN provides good load balance by distributing per-message overhead in CHAIN-like trees, nodes have to pay high cost for relaying, which seriously degrades the performance of CHAIN when workloads are heavy (performs the best when workloads are light as indicated by the left portions of both Figure 7(a) and 7(b)). MAX_AVB scheme outperforms both STAR and CHAIN given small workload, as it avoids over stretching a tree in breadth or height by growing trees from nodes with the most available capacity. However, its performance quickly degrades with increasing workload as a result of relaying cost.

Real-world performance. To evaluate the performance of REMO in a real world application. we measure the average percentage error of received attribute values for synthetically generated monitoring tasks. Specifically, we measure average percentage error between the snapshot of values observed by our scheme and compare it to the snapshot of "actual" values (that can be obtained by combining local log files at the end of the experiment). Figures 8(a) compares the achieved percentage error between different partition

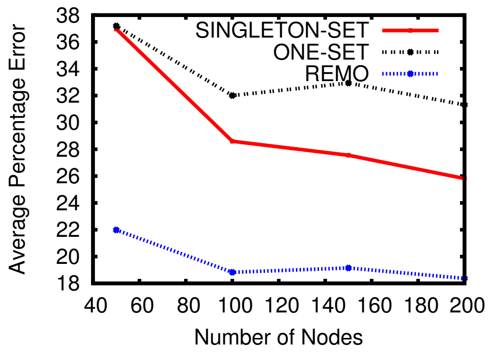


(a) Increasing $|A_t|$

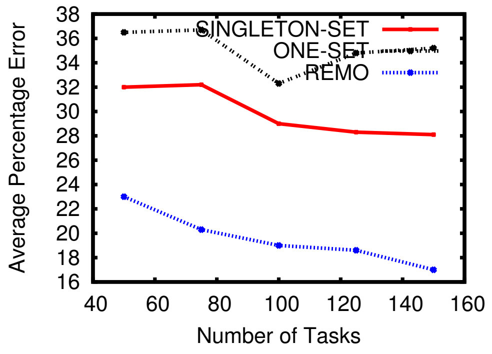


(b) Increasing Small-scale Tasks

Figure 7: Comparison of Tree Construction Schemes under Different Workload and System Characteristics



(a) Increasing Workload



(b) Increasing Tasks

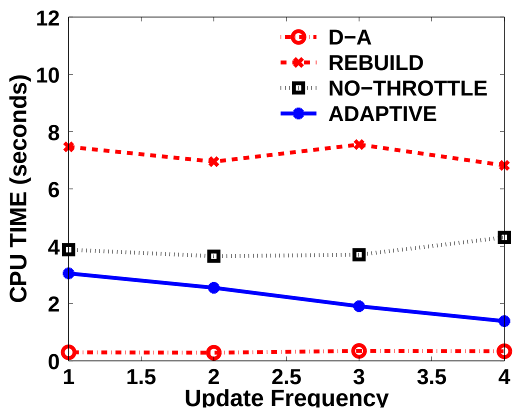
Figure 8: Comparison of Average Percentage Error

schemes given increasing number of nodes. Recall that our system can deploy the application over any number of nodes. The figure shows that our partition augmentation scheme in REMO outperforms the other partition schemes. The percentage error achieved by REMO is around 30%-50% less than that achieved by SINGLETON-SET and ONE-SET. Interestingly, the percentage error achieved by REMO clearly reduces when the number of nodes in the system increases. However, according to our previous results, the number of nodes has little impact on the coverage of collected attributes. The reason is that as the number of nodes increases, monitoring tasks are more sparsely distributed among nodes. Thus, each message is relatively small and each node can have more children. As a result, the monitoring trees constructed by our schemes are "bushier", which in turn reduces the percentage error caused by latency. Similarly, we can see that REMO gains significant error reduction compared with the other two schemes in Figure 8(b) where we compare the performance of different partition schemes under increasing monitoring tasks.

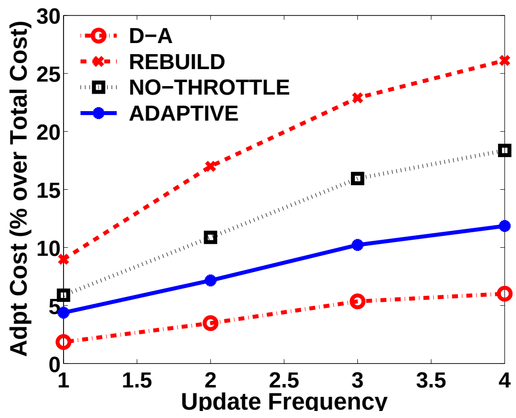
Runtime Adaptation. To emulate a dynamic monitoring environment with a small portion of changing tasks, we continuously update (modify) the set of running tasks with increasing update frequency. Specifically, we randomly select 5% of monitoring nodes and replaces 50% of their monitoring attributes. We also vary the frequency of task updates to evaluate the effectiveness of our adaptive techniques.

We compare the performance and cost of four different schemes: 1) DIRECT-APPLY (D-A) scheme which directly applies the changes in the monitoring task to the monitoring topology. 2) REBUILD scheme which always performs full-scale search from the initial topology with techniques we introduced in Section 6.3. 3) NO-THROTTLE scheme which searches for optimized topology that is close to the current one with techniques we introduced in Section 2.4. 4) ADAPTIVE scheme is the complete technique set described in Section 2.4, which improves NO-THROTTLE by applying cost-benefit throttling to avoid frequent topology adaptation when tasks are frequently updated.

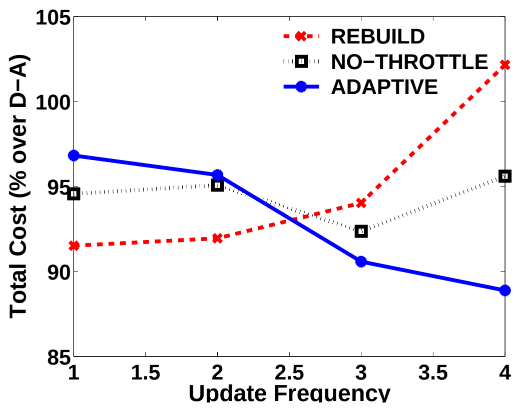
Figure 9(a) shows the CPU time of running different planning schemes given increasing



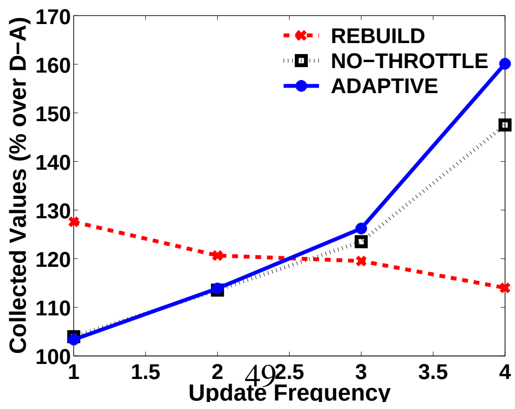
(a) CPU Time Consumption



(b) Adaptation Cost



(c) Total Cost



(d) Percentage of Collected Values

Figure 9: Performance Comparison of Different Adaptation Schemes Given Increasing Task Updating Frequencies

task updating frequency. The X-axis shows the number of task update batches within a time window of 10 value updates. The Y-axis shows the CPU time (measured on a Intel CORE Duo 2 2.26GHz CPU) consumed by each scheme. We can see that D-A takes less than 1 second to finish since it performs only a single round tree construction. REBUILD consumes the most CPU time among all schemes as it always explores the entire searching space. When cost-benefit throttling is not applied the NO-THROTTLE scheme consumes less CPU time. However, its CPU time grows with task update frequency which is not desirable for large-scale monitoring. With throttling, the adaptive schemeincurs even less CPU time (1-3s) as it avoids unnecessary topology optimization for frequent updates. Note that while the CPU time consumed by ADAPTIVE is higher than that of D-A, it is fairly acceptable.

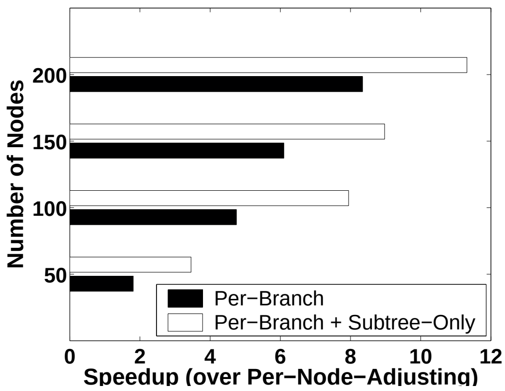
Figure 9(b) illustrates the percentage of adaptation cost over the total cost for each scheme. Here the adaptation cost is measured by the total number of messages used to notifying monitoring nodes to change monitoring topology (e.g. one such message may inform a node to disconnect from its current parent node and connect to another parent node). Similarly, the total cost of a scheme is the total number of messages the scheme used for both adaptation and delivering monitoring data. REBUILD introduces the highest adaptation cost because it always pursues the optimal topology which can be quite different from the current one. Similar to what we observed in Figure 9(a), NO-THROTTLE achieves much lower adaption cost compared with REBUILD does. ADAPTIVE further reduces adaptation cost which is very close to that of D-A, because cost-benefit throttling allows it to avoid unnecessary topology optimization when task updating frequency grows.

Figure 9(c) shows the scheme-wise difference of the total cost (including both adaptation and data delivery messages). The Y-axis shows the ratio (percentage) of total cost associated with one scheme over that associated with D-A. REBUILD initially outperforms D-A as it produces optimized topology which in turn saves monitoring communication cost. Nevertheless, as task updating frequency increases, the communication cost of adaptation messages generated by REBUILD increases quickly, and eventually the extra cost in adaptation surpasses the monitoring communication cost it saves. NO-THROTTLE shows similar growth of total cost with increasing task updating frequency. ADAPTIVE, however, consistently outperforms D-A due to its ability to avoid unnecessary optimization.

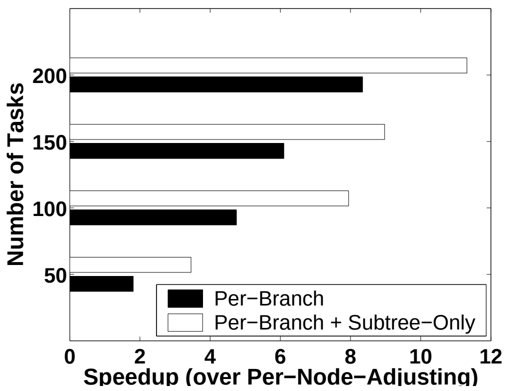
Figure 9(d) shows the performance of schemes in terms of collected monitoring attribute values. The Y-axis shows the percentage of collected values of one scheme over that of D-A. Note that the result we show in Figure 9(c) is the generated traffic volume. As each node cannot process traffics beyond its capacity, the more traffic generated, the more likely we observe miss-collected values. With increasing task updating frequency, the performance of REBUILD degrades faster than that of D-A due to its quickly growing cost in topology optimization (see Figure 9(b) and 9(c)). On the contrary, both NO-THROTTLE and ADAPTIVE gain an increasing performance advantage over D-A. This is because the monitoring topology can still be optimized with relatively low adaptation cost with NO-THROTTLE and ADAPTIVE, but continuously degrades with D-A, especially with high task updating frequency.

Overall, ADAPTIVE produces monitoring topologies with the best value collection performance (Figure 9(d)), which is the ultimate goal of monitoring topology planning. It achieves this by minimizing the overall cost of the topology (Figure 9(c)) by only adopting adaptations whose gain outweighs cost. Its searching time and adaptation cost, although slighter higher than schemes such as D-A, is fairly small for all practical purposes.

Optimization. Figure 10(a) and 10(b) show the speedup of our optimization techniques for the monitoring tree adjustment procedure, where the Y-axis shows the speedup of one technique over the basic adjustment procedure, i.e. the ratio between CPU time of the basic adjustment procedure over that of an optimized procedure. Because the basic adjustment procedure reattaches a branch by first breaking up the branch into individual nodes and performing a per-node-based reattaching, it takes considerably more CPU time compared

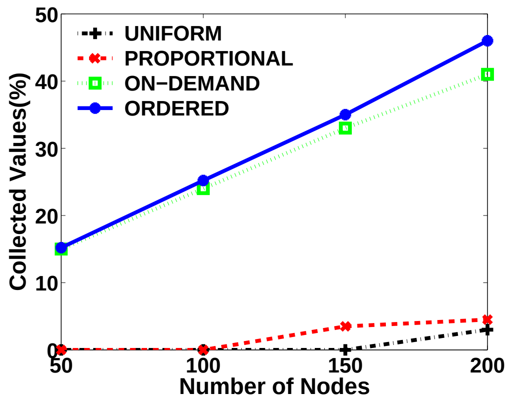


(a) Increasing Nodes

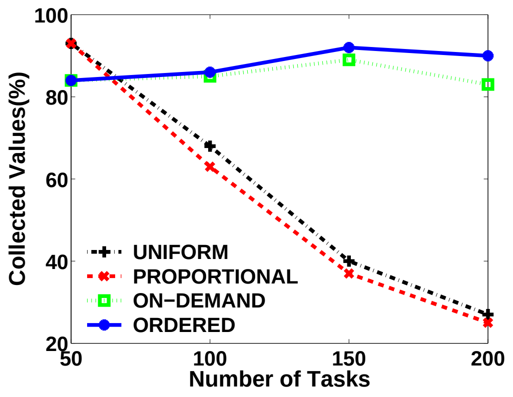


(b) Increasing Tasks

Figure 10: Speedup of Optimization Schemes



(a) Increasing Nodes



(b) Increasing Tasks

Figure 11: Comparison between Resource Allocation Schemes

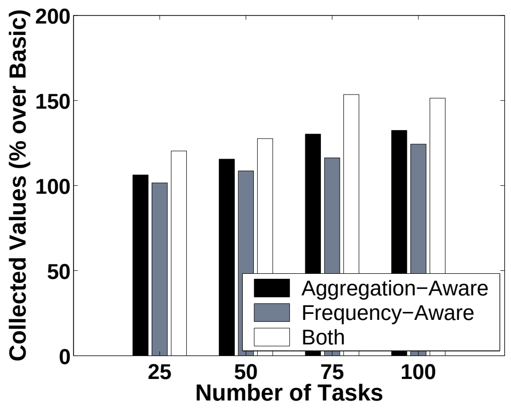
with our branch-based reattach and subtree-only reattach techniques. With both techniques combined, we observe a speedup at up to 11 times, which is especially important for large distributed systems. We also find that these two optimization techniques introduce little performance penalties in terms of the percentage of values collected from the resulting monitoring topology($< 2\%$).

Figure 11(a) and 11(b) compare the performance of different tree-wise capacity allocation schemes, where UNIFORM divides the capacity of one node equally among all trees it participates in, PROPORTIONAL divides the capacity proportionally according to the size of each tree, ON-DEMAND and ORDERED are our allocation techniques introduced in Section 2.5.2. We can see that both ON-DEMAND and ORDERED consistently outperform UNIFORM and PROPORTIONAL. Furthermore, ORDERED gains an increasing advantage over ON-DEMAND with growing nodes and tasks. This is because large number of nodes and tasks cause one node to participate into trees with very different sizes, where ordered allocation is useful for avoiding improper node placement, e.g. putting one node as root in one tree (consuming much of its capacity) while it still needs to participate in other trees.

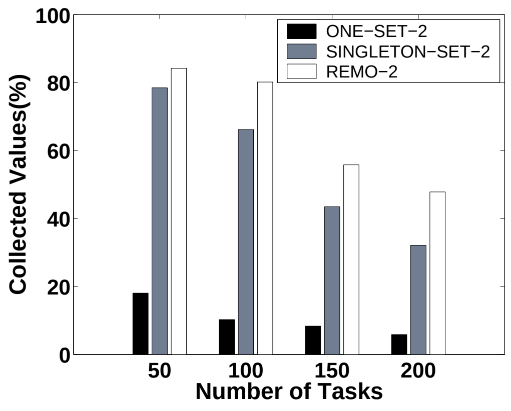
Extension. Figure 12(a) compares the efficiency of basic REMO with that of extended REMO when given tasks that involves in-network aggregation and heterogeneous update frequencies. Specifically, we apply MAX in-network aggregation to tasks so that one node only needs to send the largest value to its parent node. In addition, we randomly choose half of the tasks and reduce their value update frequency by half. The Y-axis shows values collected by REMO enhanced with one extension technique, normalized by values collected by the basic REMO. Note that collected values for MAX in-network aggregation refer to values included in the MAX aggregation, and are not necessarily collected by the root node.

The basic REMO approach is oblivious to in-network aggregation. Hence, it tends to overestimate communication cost of the monitoring topology, and prefers SINGLETON-SET-like topology where each tree delivers one or few attributes. As we mentioned earlier, such topologies introduce high per-message overhead. On the contrary, REMO with aggregation-awareness employs funnel functions to correctly estimate communication cost and produces more efficient monitoring topology. We observe similar results between the basic REMO and REMO with update-frequency-awareness. When both extension techniques are combined, they can provide an improvement close to 50% in terms of collected values.

Figure 12(b) compares the efficiency of REMO with replication support and two alternative techniques. The SINGLETON-SET-2 scheme uses two SINGLETON-SET trees to deliver values of one attribute separately. The ONE-SET-2 scheme creates two ONE-SET trees, each of which connects all nodes and delivers values of all attributes separately.



(a) Advanced Tasks



(b) Replication Support

Figure 12: Performance of Extension Techniques

REMO-2 is our Same-Source-Different-Path technique with a replication factor of 2, i.e. values of each attribute are delivered through two different trees. Compared with the two alternative schemes, REMO-2 achieves both replication and efficiency by combining multiple attributes into one tree to reduce per-message overhead. As a result, it outperforms both alternatives consistently given increasing monitoring tasks.

2.8 Related Work

Much of the early work addressing the design of distributed query systems mainly focuses on executing single queries efficiently. As the focus of our work is to support multiple queries, we omit discussing these work. Research on processing multiple queries on a centralized data stream [125, 64, 68, 72] is not directly related with our work either, as the context of our work is distributed streaming where the number of messages exchanged between the nodes is of concern.

A large body of work studies query optimization and processing for distributed databases (see [63] for a survey). Although our problem bears a superficial resemblance to these distributed query optimization problems, our problem is fundamentally different since in our problem individual nodes are capacity constrained. There are also much work on multi-query optimization techniques for continuous aggregation queries over physically distributed data streams [72, 51, 100, 122, 125, 64, 68, 22]. These schemes assume that the routing trees are provided as part of the input. In our setting where we are able to choose from many possible routing trees, solving the joint problem of optimizing *resource-constrained* routing tree construction *and* multi-task optimization provides significant benefit over solving only one of these problems in isolation as evidenced by our experimental results.

Several work studies efficient data collection mechanisms. CONCH [98] builds a spanning forest with minimal monitoring costs for continuously collecting readings from a sensor network by utilizing temporal and spatial suppression. However, it does not consider the resource limitation at each node and per-message overhead as we did, which may limit its applicability in real world applications. PIER [50] suggests using distinct routing trees for each query in the system, in order to balance the network load, which is essentially the SINGLETON-SET scheme we discussed. This scheme, though achieves the best load balance, may cause significant communication cost on per-message overhead.

Most recently, Wang, Kutare and et. al. [118, 66] proposed a flexible architecture that enables the tradeoff between monitoring/analysis costs and the benefits of monitoring/analysis results for web application performance analysis and virtual machine clustering. The architecture utilizes reconfigurable software overlays (Distributed Computation Graphs (DCGs)) which undertakes monitoring data collection, exchange and processing. While this work considers monitoring cost in terms of capital cost of dedicated monitoring hardware or software, our approach considers primarily CPU resource consumption related to monitoring communication or data collection. Furthermore DCGs focus on designing a flexible monitoring/analysis architecture. In contrast, we aim at developing concrete distributed monitoring algorithms that minimizes monitoring communication or data collection for a specific form of monitoring (state monitoring).

CHAPTER III

A SELF-SCALING MANAGEMENT SYSTEM FOR VIRTUALIZED CLOUD DATACENTERS

3.1 Introduction

Datacenter virtualization has attracted great attention in recent years due to various benefits it offers. It saves total cost of ownership by consolidating virtual servers[114] and virtual desktops[115]. It also provides high availability to applications that are not designed with this feature through live state replication at the virtual machine level[113]. Live virtual machine migration[112, 30] not only unifies servers of a datacenter into a single resource pool, but also saves power[110] during non-peak hours by consolidating virtual machines into few servers and shutting down the rest. More recently, virtualization also enables the proliferation of cloud computing platforms such as Amazon's EC2[12] where virtual machines in a datacenter can be rent based on usage.

Most virtualized datacenters today rely on management middleware to perform routine operations. In general, these middleware systems[75, 59, 28, 7, 116, 6, 5] provide unified management of physical hosts, virtual machines as well as other manageable entities such as network storage in the datacenter infrastructure. They continuously monitor all manageable entities to provide up-to-date datacenter runtime status[83, 79]. They also provide a wide range of functions such as task execution, configuration management and policy enforcement by performing complex control logic, maintaining persistent state and scheduling management jobs on individual manageable entities. Cloud applications often rely on these functions for various system supports such as resource provisioning, load balancing, performance tuning, etc. Take load balancing for example, the management system can perform live VM migration to move VMs from heavily loaded hosts to lightly loaded ones so that user applications inside VMs do not suffer performance degradation[30]. Hence, the performance of the management system has a direct impact on cloud application performance and cloud user satisfaction[101, 81].

However, while there has been a large body of work on the performance of individual virtualized hosts, the performance of management systems has been given much less attention. In fact, there are two trends suggesting that virtualized datacenter management will become increasingly critical. First, the number of manageable entities grows one to two orders of magnitude when virtualizing a physical datacenter, as each physical host runs from a few tens to hundreds of virtual machines that are directly manageable. This number will continue to grow in the coming many-core era as a single host will be able to support more and more virtual machines. Second, many virtualized datacenters are multi-tenant. These datacenters provide public or private cloud to an increasing number of individual or organizational customers who have entirely different yet overlapping requirements in cloud management and maintenance.

These two trends significantly increase the management workload in virtualized datacenters and intensify peak workloads due to multi-tenancy. Our observation[101] based on production datacenter management traces suggests that management workloads in virtualized datacenters tend to be bursty, with extremely high peak workload. Such workloads can cause considerable latencies in management task execution, which eventually hurts cloud application performance. In the load balancing example we mentioned earlier, user applications running on an overloaded host may experience performance issues if the corresponding virtual machines can not be migrated to lightly loaded hosts in a timely manner. The management system therefore should minimize the impact of the bursty workloads.

However, finding an efficient and effective solution to this issue is challenging, because provisioning the management system based on regular workloads or peak workloads would either introduce high task execution latency during peak workloads, or make significant datacenter resources unavailable to customers, which eventually raises cost of ownership and causes resource contention. We argue that one possible solution to efficiently handle management workload bursts is to make the management system self-scaling, which allows the management system to automatically boost its capacity during peak workloads.

In this chapter, we introduce Tide, a prototype management system with dynamic capacity that scales with management workloads. Tide uses VMware's vSphere management server as building blocks. It consists of a primary management server and a number of virtual management server instances that are powered off during normal workloads. To deal with delayed task execution caused by bursty workloads, Tide can dynamically power on virtual management server instances to boost overall throughput.

Self-scaling has two fundamental requirements. First, management instances provisioning should be fast, and the number of provisioned instances should be reasonably small. Second, provisioned instances should be fully utilized to maximize the overall task execution throughput. Tide employs two sets of novel techniques to meet these requirements on two different levels. On the instance provision level, we devise a novel provisioning algorithm that can *quickly* provision *just enough* server instances to the management system for the current workload. The algorithm considers the management system as a black-box and requires no performance modeling of server instances or workload profiling. We apply formal analysis to show the effectiveness of this algorithm. In addition, we also introduce two optimization techniques that further reduce latencies of the provision process. On the workload dispatching level, we propose a simple yet effective scheme for workload dispatching which maximizes the utilization of instances by avoiding costly task rebalancing among instances.

To the best of our knowledge, Tide is the first work dedicated to the design and implementation of the self-scaling feature in virtualized datacenters management systems. Compared with recent work on flexible monitoring/management architecture [66] that adopts a clean slate approach, we build Tide with VMware's vSphere server, a widely used commerical virtualized datacenter management system. Using multiple vSphere servers as a dynamic component, however, brings several challenges. Through careful implementation, we address problems such as long instance startup time and consistency in distributed task execution. These low level techniques are critical to the performance of Tide. To evaluate the effectiveness of Tide, we perform extensive experiments with both synthetic and real world management workload collected from several virtualized datacenters. Our experiment results show that Tide can quickly react to workload bursts and minimize task execution latencies caused by workload bursts. It also consumes much less resource compared with systems implemented with simple alternative provisioning and workload dispatching techniques.

The rest of the chapter is organized as follows. Section 3.2 introduces the background of virtualization management systems. Section 3.3 presents an overview of Tide. We describe the provisioning algorithm of Tide in Section 3.4, and introduce workload dispatching techniques in Section 3.5. We describe the implementation details of Tide in Section 3.6. We discuss important features such as fault tolerance and state consistency of Tide in Section 3.7. Section 4.7 presents our experiment results. We discuss related work in Section 6.5.

3.2 Background

Virtualization has been a major enabler of cloud computing. Many commercially-available cloud computing solutions are based on virtual machines running in datacenters[12, 106]. These datacenters may contain thousands of servers and tens of thousands of virtual machines, and these numbers will continue to grow with increasing cloud users. Managing these servers and their virtual machines individually with low-level tools is difficult. Hence, management middlewares are designed and built for centralized and efficient management of virtualized datacenters. We next introduce virtualized datacenter management systems based on a representative system vSphere from VMware. We also discuss other systems in Section 6.5.

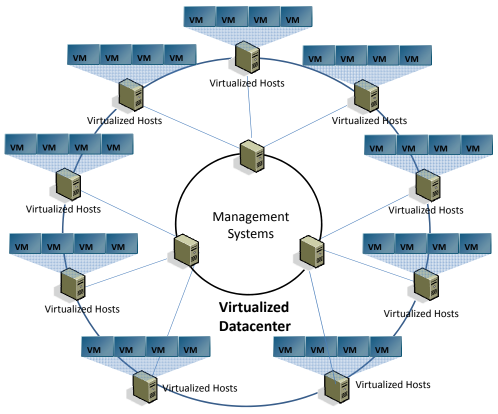


Figure 13: A High Level View of A Virtualized Datacenter

3.2.1 vSphere Datacenter Management System

Figure 13 shows a high level view of a virtualized datacenter. vSphere provides three key management functionalities. First, it executes various manual or automatic management tasks. For example, an administrator may create 100 virtual machines for new remote users. vSphere itself may continuously balance server workload via VM live migration[30]. Second, it maintains a management connection to each virtualized host(ESX host). Through this connection, it monitors the runtime status of each physical server as well as virtual machines hosted on it. Third, it maintains configuration information of the datacenter such as detailed hardware information, virtual machine and virtual network configuration. It also enforces various management policies across the datacenter. This focus of this study is on the performance of management task execution.

vSphere can execute multiple tasks at the same time. It has a pool of task-execution threads. When receives a management task, it picks one thread to execute the task if at least one thread is available. If all threads are busy, it puts the task in a FIFO queue and waits for threads to become available.

Executing management tasks often causes intensive resource consumption on the management system due to the complexity of virtualization, configuration and datacenter topology. Consider, for example, a task that selects the optimal host to power on a virtual machine[110]. It demands a series of computation involving load examining on hosts and determining the compatibility matrix between different VMs and hosts. The CPU and memory cost of such computation grows with the number of hosts and VMs. Performing such a computation for a large number of VMs at once can be quite resource-intensive.

3.2.2 Management Workloads

From management traces collected from several virtualized datacenters, we observe that management workloads in virtualized datacenters tend to be bursty[101]. Figure 14 shows the CDF of management task rates within a 10-minute time window for one datacenter trace. While the average workload is fairly small, the peak workload can be nearly 200 times higher than the average workload. There are many examples of such bursty workloads. Companies using massive virtual machines as remote desktops often perform large amount of virtual machine cloning and reconfiguration tasks during a short time window. Furthermore, they also generate a large amount of virtual machine powering on operations when employees start to work in the morning. There are many other examples, such as large-scale virtual machine provisioning for software-as-a-service, live migration based datacenter-scale load balancing, massive security patching, etc. In general, the burstiness in management workloads is inherited from the massive use of virtual machines and automatic management. We also expect this workload burstiness to grow in the near future, because the increase of cloud users eventually leads to the growing of virtual machines in datacenters, and the use of many-core will also significantly increase the number of virtual machines running on a single host.

When receives a large number of tasks during a workload burst, the management system often has to put most of the tasks in the waiting queue, which introduces considerable latency to task execution. Unfortunately, execution delays can cause various availability and performance issues to applications running in the datacenter. For example, if a

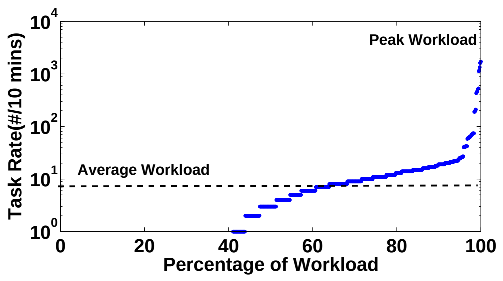


Figure 14: Burstiness of Management Workload

customer needs to spawn hundreds of virtual web servers within a short time period to accommodate flash crowd[13], delay of VM creation would very likely to cause performance degradation and even SLA violation to the customer application. Similarly, failing to perform live-migration-based load balance on time would also introduce resource contention to customer applications. Therefore, we believe the bursty workload issue is an urgent problem for the management system.

3.3 Tide Overview

We argue that one possible way to address the bursty workload problem in virtualized datacenters is to allow the management system to dynamically provision new management instances to itself during workload bursts. We refer to such a feature as self-scaling. Self-scaling has the potential to significantly reduce task execution latencies during workload bursts. Consider, for example, powering on 3000 VMs within a customer-specified 10-minute maintenance window. As VM power on tasks involve compatibility and placement computations, such workload can easily saturate a single management server. But if we can dynamically provision additional management instances, we can parallelize task execution and minimize execution latencies. Compared with static provisioning based on peak workload (e.g. always keep 30 management servers running instead of 1), self-scaling provisions resources only at peak workloads, and thus, can save considerable resources as well as maintenance cost in the long run. Following this idea, we develop Tide, a prototype self-scaling management middleware for virtualized datacenters.

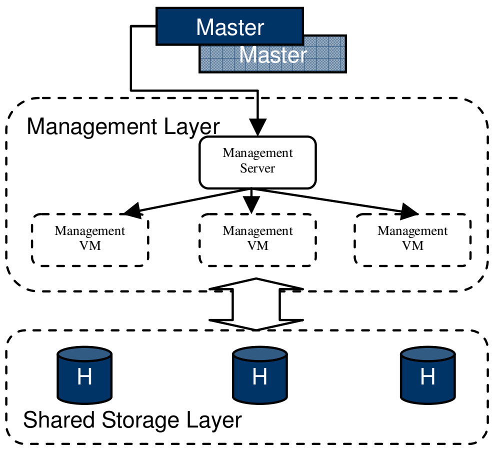


Figure 15: Block Diagram of our Design. The master nodes do auto-scaling of the management layer, which consists of vSphere VMs that store data in the global persistence layer. The managed hosts and application VMs are not shown in this diagram, but communicate with the management server/management VMs.

3.3.1 The Architecture of Tide

Figure 15 illustrates the high level architecture of Tide. In our prototype, we split the task of management into 3 software components: master nodes, the management layer and the shared-storage layer. The basic principles behind the design of Tide are as follows. First, we want to create a management layer that is small but expandable. This allows Tide to occupy small footprint of resources for normal management workloads and dynamically increase its capacity only during bursty workloads. Second, we want to make the management nodes as stateless as possible. This is important for building a expandable management layer with low dynamic provisioning complexity and fast provisioning speed. Third, separating management nodes from the data they manipulate is necessary. Management related data should be stored in a globally-accessible storage layer so that distributed management nodes share the same global view of the entire datacenter. Finally, when scaling out, the management layer can be scaled independently from the storage layer. This means that we do not pay the price of allocating storage just by virtue of creating new management nodes. We present implementation details of these three components in Section 3.6.

Master Nodes. Tide employs a set of master nodes to present a single well-known IP address to clients. A master node decides when to increase or decrease management nodes at the management layer. It also creates a mapping between management nodes and the hosts that they are managing. When a client needs to perform an operation on a given host, it first communicates with the master node to determine which management node to speak to. The client then caches this mapping so it can communicate directly with the appropriate management node on subsequent requests. The cache is invalidated as required if a host is moved. Due to the importance of the management node, master nodes can be replicated for high availability. However, the system is still functional even with the temporary absence of the master nodes.

The Management Layer consists of the collection of management nodes that are managing the entire infrastructure. Each management node is a virtual machine running vSphere (modified version), VMware's virtualized datacenter management server. This set of VMs provides equivalent functionality to a single centralized vSphere server or a Linked-vSphere server[111]. Rather than providing a single vSphere server or a small set of vSphere servers and asking the user to statically partition the hosts among the vSphere servers, we automatically determine the number of vSphere servers necessary to manage the infrastructure for good performance. For the rest of the chapter, we use the term management nodes, management instances and management VMs interchangeable. As the infrastructure grows or as the management workload increases, the management VMs automatically spawn more management VMs to accommodate the added management load. Under the coordination of the master node, these management VMs automatically repartition the physical hosts to provide scalable management. As the management workload decreases, the master node can suspend or remove these additional management VMs and again repartition the hosts among the remaining managers.

We place the vSphere management server within VMs instead of physical machines for several reasons. First, VMs can easily be created and destroyed on-the-fly. This allows our approach to easily scale with the management load. Furthermore, virtual machines can achieve near-native performance, due to recent processors with hardware support for virtualization and advances in hypervisor software. Second, the VMs can easily be moved in case of node failure, and can be placed in a preconfigured, suspended state for quick resumption in case of management load.

The Shared Storage Layer consists of an implementation of a globally-shared HBase[17], an open source incarnation of Google's Bigtable[26]. In a typical vSphere installation, each vSphere server is connected to its own database. When a managed object like a host must be moved from one vSphere server to another, the object's information must be read out of the first vSphere's database and then stored in the second vSphere's database. In contrast, with a globally-shared database, it is faster to move managed objects between vSphere instances because the data does not have to be moved: both instances can share the same backend data. With globally-shared semantics, it is also easier to implement replication of data for high availability as well as the potential for caching across geographies for spatial locality of requests. In the current implementation of Tide, the shared storage layer is statically provisioned. As the shared storage layer rarely becomes bottlenecks, we simply use a static deployment regardless of the size of the management layer. We leave dynamic provisioning of the shared storage layer, a.k.a. self-scaling of the storage layer, as our future work.

3.3.2 Challenges of Delivering Self-Scaling

Delivering self-scaling has two fundamental challenges. First, *self-scaling should provision an appropriate number of virtual instances based on received workload in a short time.* Provisioning speed and efficiency are both important because they determine task execution delay and the overall management cost. Achieving this is nontrivial as workloads and virtual instance performance are difficult to model. Second, *self-scaling requires all virtual instances to be fully-utilized for maximizing throughput.* The solution to this goal is also not straightforward as it involves scheduling of distributed virtual instances. In the rest of the chapter, we discuss our approach for these two challenges in detail.

3.4 Fast and Efficient Provisioning

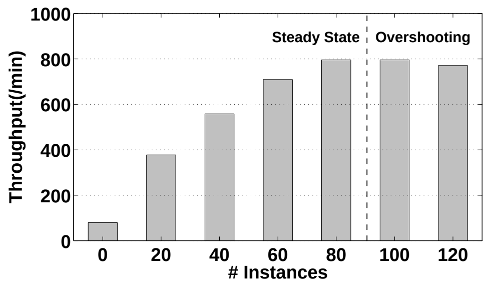
Directly estimating the appropriate number of management instances for a given workload is difficult. First of all, the resource consumption of management tasks is hard to predict. Different tasks consume CPU and IO quite differently. E.g., a VM power-on task involves computation for choosing the best host to run a VM, while cloning a VM is disk-intensive. Even tasks of the same type vary heavily in execution cost due to task heterogeneity. Second, the performance of a management instance is also different when it runs on different hosts(e.g. different CPUs). Capturing these uncertainties with performance modeling is not only hard but also makes provisioning system-dependent.

One possible solution is to use iterative schemes which repeatedly add new instances into the system until doing so does not improve throughput. For such schemes, one difficulty is to determine the number of instances to add at each iteration, a.k.a *the step size*, and straightforward schemes often do not work well. For instance, A scheme that adds a constant number k of instances each time faces a *speed-efficiency* dilemma. If k is small, the scheme may take a long time to provision sufficient instances. If k is large, it may unnecessarily provision a large amount of instances for small workload bursts, which causes resource contention between management and applications.

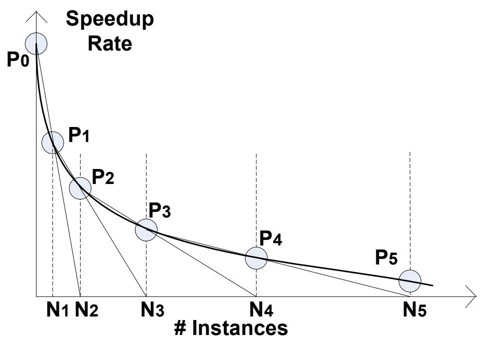
In Tide, we devise a novel adaptive approach that can *quickly* provision *just enough* instances for a workload burst. It is system-independent as it does not rely on specific workload or system performance models. The key technique in our approach is monitoring throughput speedup to decide the right amount of instances to add in the next iteration. We also show that our approach has guaranteed convergence property even under varying workloads. We next present details of this approach.

3.4.1 Speedup-Guided Provisioning

Figure 16(a) shows the relations between the number of management instances, denoted by N, and the task execution throughput. Here, we ran a workload trace collected from a production datacenter on Tide multiple times, with an increasing number of management instances each time. More details on setup can be found in Section 4.7.



(a)



(b)

Figure 16: Instance Provisioning: (a)Throughput Improvement with Increasing Instances; (b)The Speedup Rate Curve

As N increases, the throughput increases to a point, where the throughput of the system matches the workload. After that, the throughput levels off even with larger N, because the gain in throughput is limited but the cost of coordinating management instances continues to grow. We refer to the throughput increment after adding one or more instances as *throughput speedup*, and the state where the throughput levels off as *steady state*. In addition, we refer to the situation of provisioning after steady state as *overshooting*. Overshooting is clearly undesirable as it wastes resources. Ideally, we should stop provisioning new instances when the system enters the steady state. The next question is, how to reach this state as quickly as possible without causing overshooting?

A key observation here is that, as the throughput approaches to steady state, the change rate of speedup in throughput decreases. Tide uses the change rate information to guide the provisioning process. It iteratively adds a certain number of instances to the management system based on previous speedup change rate. This feature allows us to quickly approximate the steady state.

We use $T(N)$ to represent the throughput given N instances. Since speedup is a function of instance number N, we use $f(N)$ to denote the speedup of N instances. Note that $f(N)$ is the throughput increment of the system from $N-1$ instances to N instances, i.e. $f(N) = T(N) - T(N-1)$. Figure 16(b) shows the curve of $f(N)$ generated from Figure 16(a). Clearly, the system enters the steady state when f reaches the X axis, i.e. $f(N_s) = 0$ where N_s is the instance number of the system at steady state. Thus, the estimation of the steady state can be formulated as a root-finding problem.

Our provisioning algorithm is based on Secant method[104] which is a widely used root-finding technique in numerical analysis. The algorithm combines the root-finding process with the provisioning process. It also inherits several good properties from Secant method as we show later in Section 3.4.2.

For the first iteration, we measure the initial speedup by adding one instance, i.e. $f(N_0)$ where $N_0 = 1$. Clearly, $f(N_0) = T(1) - T(0)$ where $T(0)$ is the throughput of the primary vSphere server. The value of $f(N_0)$ is shown by the P_0 in Figure 16(b). Similarly, we then add a fixed small amount of instances to make the total instance number to N_1 and measure the speedup at N_1, i.e. $f(N_1) = T(N_1) - T(N_1 - 1)$, as shown by the point P_1. Based on P_0 and P_1, we find the number of instance to provision in the next iteration, N_2 as follows. We generate a linear function $S = g(N)$ which passes the point P_0 and P_1. The root of $g(N)$, as shown by N_2 in Figure 16(b), is the number of instances to provision in the next iteration. Formally, given $N_i, N_{i-1}, f(N_i), f(N_{i-1})$, we generate the number of instances to provision in $(i+1)$-th iteration as follows,

$$N_{i+1} = N_i - \frac{N_i - N_{i-1}}{f(N_i) - f(N_{i-1})} f(N_i) \tag{2}$$

The provisioning process repeats the above iteration by using two previous provisioning points to determine the next one. It terminates when the speedup improvement rate between the two most recent provisioning is below a pre-defined threshold γ. Note that users can set γ based on their performance requirements on Tide and the amount of resources they can assign to Tide. In addition, we measure throughput based on multiple samples to avoid incorrect step size estimation due to unstable readings.

To make our algorithm robust, we also apply three restrictions to our algorithm to prevent faulty provisioning.

Restriction 1 $N_{i+1} > N_i$ if $f(N_i) > 0$ ensure N_i is a increasing series (dealing with workload fluctuation)

Restriction 2 When $f(N_i) = 0$, gradually decreasing N_i (overshoot preventing)

Restriction 3 Ensure $N_{i+1} < N_i + m$, where m is the maximum step size (dealing with the divide-by-zero problem)

Note that restriction 2 is also the shrinking process in which Tide reduces its management instances when workload bursts disappear. We show later in Section 3.4.2 that these restrictions allow our algorithm to work even under varying workload.

The speedup-guided provisioning algorithm takes advantage of the quantity information in speedup to better approximate the steady state. As an example, if recent provisioning leads to similar speedup, the algorithm would provision much more instances in the next iteration (because the generated linear function $g(N)$ has relatively small tangent and, accordingly, has a relatively large root). Clearly, this feature is desirable as the system can quickly approach steady state with more instances when the performance is more predictable.

3.4.2 Performance Analysis

Overshooting in provisioning can cause inefficiency resource consumption and resource competition between Tide and applications running in datacenters. The following theorem shows that our provisioning algorithm is free from overshooting problem given non-decreasing workload.

Theorem 2 *Given that the received workload does not decrease during the provisioning process, the speedup-guided provisioning algorithm does not cause overshoots.*

Proof 1 *Given non-decreasing workload, the speed up rate curve is a convex function in which the speedup rate drops slower as the number of instances added. The number of instances to provision determined by Equation 1 is bounded from right by the root. Therefore, N_i generated from the Secant solver does not exceed the root. In other words, N_i can not cause an overshoot.*

Provisioning speed is also an important concern as slow provisioning can cause delayed task execution. The following theorem shows that our approach can quickly converge to the steady state.

Theorem 3 *Given that the received workload is stable, the number of instances in Tide super-linearly converges to the number of instances in the steady state.*

Proof 2 *Suppose N_i converges to N_m, there exists two positive constants, A and $R > 1$ such that $\lim_{i\to\infty}(N_m - N_{i+1})/(N_m - N_i)^R = A$ Then N_i is said to converge super-linearly with the order of convergence R. Secant method is widely known to have the order of convergence equals to the golden ratio (1.618), which is slightly smaller than the order of convergence of the Newton's method but still larger than one. (We didn't use Newton's method as it requires accurate estimation of the curve's second order.) Thus, Tide super-linearly converges to the ideal instance number in the steady state.*

Real world workload often varies over time. An ideal provisioning algorithm should still be able to provision correctly even when the incoming workload is not stable. The next theorem shows that the speedup-guided algorithm has this feature. Our experiment results also show that the proposed algorithm can quickly approaches steady state under real world varying workloads.

Theorem 4 *The proposed provisioning algorithm can eventually provision the right number of instances, even if the workload varies during the provisioning process.*

Proof 3 *Suppose before the workload changes we have computed the speedup information $f(N_i)$ and the next instance provision number N_{i+1}. As the workload varies, the speedup curve changes from f to f' and the ideal provisioning number is also changed. We prove the theorem from the following cases.*

Case 1: the workload increases and $f'(N_{i+1}) < f(N_i)$. The next provisioning number N_{i+2} might be still smaller than the ideal number. We can always tell if this is true by $f'(N_{i+2})$. If true, we continue Secant solving process(Equation 1) until convergence. The information prior to the workload change imposes no effect on the subsequent computation. If not, then by Restriction 2, the provision number is gradually decreased until it reaches the ideal value.

Case 2: the workload increases and $f'(N_{i+1}) > f(N_i)$. Using Secant method to compute N_{i+2} from $f'(N_{i+1})$, $f(N_i)$, N_{i+1} and N_i give us a smaller result. By restriction 1, this result is invalid if $f'(N_{i+2})$ is still larger than zero. Thus, the algorithm will set N_{i+2} to $1 + N_{i+1}$ before assuming Secant solver for the next provisioning number. Therefore the information prior to the workload change becomes obsolete.

Case 3: the workload decreases and $f'(N_{i+1}) = 0$. In this case, an overshoot might have already happened. By Restriction 2, the provision is gradually decreased.

Case 4: the workload decreases and $f'(N_{i+1}) > 0$. Because we assume the workload decreases, $f'(N_{i+1})$ can only be less than $f(N_{i+1})$, i.e. the speedup is not as large as it was before the workload decrease. In this case, the ideal provision number has not been *reached yet, so it is safe to continue the secant solver without triggering the overshoot preventing mechanism.*

The preceding discussion covers all cases when the proposed algorithm confronts a workload fluctuation. We prove the robustness by showing that the right number of instances can always be reached.

3.4.3 Optimization

Reducing Measurement Latencies. In the original algorithm, since we need to measure both $T(N)$ and $T(N-1)$ to get $f(N)$, the algorithm measures throughput twice for each iteration. This, however, limits provision speed as measurement takes time. We later find that it is possible to improve this algorithm by performing only one measurement for each iteration. Specifically, instead of calculating the speedup by $f(N_i) = T(N_i) - T(N_i - 1)$, we let $f(N_i) = \overline{f}(N_i, N_{i-1}) = (T(N_i) - T(N_{i-1}))/(N_i - N_{i-1})$. As a result, we can approximate the speedup of the i-th iteration based on measured T_{N_i} and previous measurement $T_{N_{i-1}}$. The following theorem shows that this optimization technique does not change the properties of the provisioning algorithm.

Theorem 5 *Theorem 1, 2 and 3 still hold when we substitute Equation 2 with the following equation:*

$$N_{i+1} = N_i - \frac{N_i - N_{i-1}}{\overline{f}(N_i, N_{i-1}) - \overline{f}(N_{i-1}, N_{i-2})} f(N_i)$$

, where $\overline{f}(N_i, N_{i-1}) = \frac{T(N_i) - T(N_{i-1})}{N_i - N_{i-1}}$.

Proof 4 *Here f is reformulated to $\overline{f}$ that allow two variables N_i and N_{i-1} which are not necessarily close to each other. However, we will prove in the following that $\overline{f}$ is still isomorphic to a single-valued function in our case. As previously discussed, $T(N)$ is monotonically increasing and $T(N)/dN$ is monotonically decreasing. By the mean value theorem, $\overline{f}(a, b) > \overline{f}(b, c)$, for any positive integers $a < b < c$. Denote $\overline{f}(N_i, N_{i-1})$ as F_i, i.e. $F_i = \overline{f}(N_i, N_{i-1})$, $F_{i+1} = \overline{f}(N_{i+1}, N_i)$. $\{F_i\}$ is a monotonic decreasing chain with* *respect to the input sequence $\{Ni\}$. Therefore, the meeting point of $\{F_i\}$ and the N_i axis can be predicted by Secant method.*

Virtual Instance Configuration. Provisioning a new virtual management instance usually involves powering on the corresponding virtual machine, loading OS and starting the vSphere management service. We refer to the time an instance takes to be ready to accept workloads as *ready time*. Our second optimization technique reduces the ready time to minimize the time one provisioning iteration takes.

We find that loading OS and starting vSphere service in a virtual machine takes considerable time(about 2-3 minutes) as vSphere is not designed for frequent and quick startup. To minimize this delay, we boot up virtual instances once after their initial creation, start its vSphere service, and then suspend these virtual machines. During self-scaling, Tide simply resumes a virtual instance when it needs one. Resuming a virtual machine means loading the memory image of the virtual machine from disks. Hence, the length of resuming depends on the memory footprint of a virtual instance. The average ready time of a 512MB-instance on a Dell PowerEdge 1850 with SCSI disks is 16 seconds, but increases quickly to 76 and 203 seconds with 1GB-instance and 2GB-instance. Hence, we set the memory size of a virtual instance to be 512MB, because the corresponding resuming process is much faster than cold booting and starting vSphere service, and the instance performs reasonably well with this much memory.

3.5 Maximizing Instance Utilization with Workload Dispatching

Driving management instances to high utilization is important for resource efficiency. If provisioned instances are under-utilized, the provisioning algorithm has to provide additional management instances, which may cause unnecessary resource competition between Tide and applications running in the datacenter. We next discuss how to achieve high utilization through workload dispatching.

3.5.1 Workload Dispatching in Tide

Tide performs workload dispatching in a per-host basis. Specifically, management tasks are associated with the corresponding virtualized hosts(ESX hosts). For example, a power-on task of virtual machine V is associated with the ESX host running V, and the ESX host may be associated with multiple tasks related with its virtual machines. During workload dispatching, Tide assigns an ESX host to a virtual instance so that the instance can execute all tasks on the ESX host. Tide employs host-based dispatching simply because each ESX host can only be managed by one vSphere server at a time.

Dispatching a host to a virtual instances requires the consideration of two factors, the available *task-execution threads* and the available *entity space*. A virtual instance has a fixed number of task-execution threads and each thread can execute one management task at a time. When all threads are busy, a virtual instance puts addition assigned tasks into a waiting queue until one thread becomes available. We use the ratio of busy threads to measure instance utilization. In addition, entity space is the maximum number of virtual machines one instance can manage simultaneously. As an instance continuously maintains both runtime status and configuration information of its managed virtual machines, the entity space is a fixed number primarily determined by its memory size and CPU capacity. Exceeding the entity space limit would lead to significant performance degradation.

Based on these two factors, ideal workload dispatching should keep high utilization on instances and avoid exceeding entity space limits at the same time. Hence, we first implemented an intuitive dispatching algorithm which eagerly reassigns hosts from overloaded instances to lightly-loaded ones without exceeding space limits. Specifically, the algorithm monitors the utilization, waiting tasks, and available space (i.e., the remaining number of virtual machines one instance can manage) on all virtual instances. Whenever it finds a host with all its tasks finished, it assigns the host back to the primary instance. This is to maximize the available space on virtual instances. In addition, if it finds an instance with available threads and entity space, it randomly assigns a host with waiting tasks to the instance so long as doing so does not violate space limit.

However, the eager reassignment algorithm does not perform well as we show in Section 4.7. There are two reasons. First, it eagerly rebalances workload among instances and causes frequent instance-host reconnection, which is both expensive and time-consuming. Second, eager workload rebalancing also minimizes the available space on instances at all times. However, workload bursts often shifts from one set of hosts to a different set. When it happens, the algorithm has to wait for virtual instances to finish all tasks associated with certain hosts and reassign the hosts back to the primary instance before it can assign other hosts with new workload to virtual instances. This causes low utilization of virtual instances.

3.5.2 An Online Dispatching Algorithm

Based on this observation, we believe a workload dispatching algorithm should *reserve* some entity space to handel workload shifts. Following this idea, we propose an online algorithm that dispatches workload based on the gain of host assignment. Specifically, we define *workload density* of host h as $\rho_h = \frac{T_h}{V_h}$ where T_h and V_h are the number of waiting tasks and the number of virtual machines on host h. Clearly, the higher ρ_h is, the more worthwhile assigning h to a virtual instance is. Given a set of hosts H with waiting tasks at any time t, the algorithm first finds the host $h \in H$ with the highest ρ_h. If $\rho_h \geqslant \mu$ where μ is a predefined gain threshold, it assigns h to a randomly selected instance i with available space $S_i \geqslant V_h$.

The next question is how to select a proper value for μ that maximizes instance utilization. In fact, finding the best value for μ can be modeled as an online knapsack problem. Consider a host h as an item with value T_h and weight V_h, and an instance $i \in I$ as a knapsack with capacity S_i where I is the set of all instances. The problem can be defined as:

Given a knapsacks with capacity $S = \sum_{i \in I} S_i$, an items h having a value T_h and weight

V_h at each instant, put h into the knapsack iff $\rho_h = \frac{T_h}{V_h} \geqslant \mu$ and $V_h \leqslant S$. Find the best value for μ so that the total value of items in the knapsack is maximized.[1]

To find the optimal μ, we use the theoretical results in [128] by setting

$$\mu = (\frac{Ue}{L})^z(\frac{L}{e})$$

, where L and U be the lower and upper bound of workload density for all hosts and and z is the fraction of space filled over all instance space in I. Note that Equation 3 allows the dispatching algorithm to dynamically decide whether to assign a host based on current workload and available space. When the available space is large ($z \to 0$), $\mu \to L$ and the algorithm always assigns hosts to lightly loaded instances. Accordingly, when $z \to 1$, $\mu \to U$ and the algorithm assigns only hosts with high workload density to virtual instances. It can be also shown that setting μ based on Equation 3 achieves a total value of at least $1/\ln(\frac{U}{L}+1)$ of the optimal assignment[128]. As we show later, the online dispatching algorithm achieves considerable higher instance utilization compared with the eager algorithm.

3.6 Prototype Implementation

Before we describe implementation details of Tide, we first explain some terminology. By infrastructure, we mean the entire set of ESX hosts and the VMs that are running on them. When we discuss vSphere VMs, we mean VMs that are running somewhere in the infrastructure and are running the vSphere service to manage other ESX hosts in that infrastructure. Note that these vSphere VMs reside within the same ESX hosts that comprise the infrastructure.

[1]We consider all instances as a single knapsack for simplicity, although one item may not fit into any instance but fits into the knapsack.

3.6.1 Master Nodes

The first task of the master nodes is to divide the ESX hosts among available vSphere servers. One way to do this is to implement an auto-discovery module such that ESX hosts, when come online, first communicate with a master node. This master node notifies the appropriate vSphere server which then adds the host into its managed host list. The master node also retains this mapping information. When the number of hosts per vSphere exceeds a pre-determined threshold, new vSphere VMs are spawned to manage the new hosts. In our design, we perform this mapping once at initial startup of the master node, and update the mapping as new hosts come online.

The second task of the master nodes is to create new vSphere VMs or remove vSphere VMs and re-distribute ESX hosts as the number of active tasks in the system increases or decreases. We chose to do this by directing all task requests through a globally-shared task queue. The master nodes can monitor this task queue and determine when more or less vSphere VMs are needed. When more vSphere VMs are required, the master node can create new nodes and assign the tasks appropriately, also redistributing ESX hosts as needed. Another possible method (not implemented here) is to modify the API to allow task stealing from vSphere servers.

In our prototype, we focused on auto-scaling the management layer when a burst of tasks has been detected. All tasks are sent to the master nodes first. The master node examines the depth of the task queue and then decides to spawn additional vSphere servers to respond to this set of tasks. Since these vSphere servers are originally in a suspended state, the master node issues commands to resume these vSphere VMs. The master node then disconnects hosts from the pre-existing vSphere servers and connects them to the newly-created vSphere servers. The add-host commands access state from the globally-shared database in order to quickly start up the vSphere servers.

As noted above, the master node must make two important decisions: it must determine how many vSphere servers to be spawned, and it must redistribute tasks and hosts/VMs appropriately. It determines how many vSphere nodes to spawn by examining the available CPU/memory capacity in the infrastructure and by examining the number of outstanding tasks to be performed. If there is minimal excess capacity, the master node spawns fewer additional vSpheres. Next, the master node examines the task queue and redistributes tasks and hosts/VMs among vSphere servers so that all vSphere servers will complete tasks at approximately the same time. Ideally, all of these decisions would be made dynamically. In practice, for our testing, we pre-spawned a modest number of vSphere VMs and put them in a suspended state so that they could be easily restarted and deployed as live vSphere servers.

3.6.2 The Management Layer

In our prototype, the management layer is implemented using standard vSphere 4.0 servers deployed in VMs. Each vSphere server is responsible for managing a subset of the ESX hosts in the infrastructure. To allow vSphere VMs to be created quickly, we pre-deploy initialized vSphere VMs in a suspended state throughout the infrastructure so that quick deployment merely requires resuming from a suspended state rather than creation and initialization. Before putting the management VMs in a suspended state, we run the vSphere process within each vSphere VM until the vSphere process has initialized itself and is ready to manage hosts, but has no hosts connected. When handing a managed host to a newly-resumed vSphere VM, we can simply ask the vSphere VM to connect to the managed host. Another option is to perform fast cloning of vSphere VMs using techniques like those described in SnowFlock[67].

The vSphere platform allows the user to divide a virtualized datacenter into clusters, which are groups of hosts with similar configurations. The cluster is a unit of admission control: before VMs are powered on, vSphere makes sure that there is sufficient capacity in the cluster to turn on the VMs. This requires some synchronization that is cluster-wide. If a cluster contains a large number of hosts and we wish to split it across vSphere instances, then there must be some way to preserve the same synchronization semantics for the cluster. For our prototype, we chose not to split clusters between management instances, and instead ensure that all of hosts and VMs for a given cluster are managed by a single vSphere instance. If it were necessary to split hosts within a cluster across management instances, we could use a distributed lock model similar to Chubby locks[25]. Basically, when an operation occurs to a host that is within a cluster, a distributed lock is required to perform the operation. For our prototype, however, we chose to confine clusters to be managed by single vSphere instance.

3.6.3 The Shared Storage Layer

The shared storage layer uses the HBase[17] implementation of Bigtable[26]. We use a Bigtable-like approach because we wanted scalability and built-in redundancy for high-availability. In HBase, data is addressed using a (row,column,version) tuple, and data is stored in a column-oriented format. Because this addressing scheme is very different from the addressing scheme used in a traditional relational database (used by vSphere), we implemented an RPC module and a translation module to allow current vSphere servers to communicate with HBase. The vSphere server issues its normal requests, which are converted by the RPC server and translation module into HBase requests. In addition, the translation module is capable of implementing various common RDBMS calls like joins in terms of HBase primitives. In this manner, the current vSphere server code did not have to be modified significantly to interact with HBase. In addition, as HBase continues to evolve and include more and more SQL-like functionality, it will become feasible to use that functionality and remove some of these translations from our translation module.

The performance of HBase depends on how data is distributed among different HBase servers. To achieve the best performance, we split tables into small table regions (both row-wise and column-wise) and distribute regions evenly across HBase servers. Using small table regions enables us to even-out the workload among different HBase servers and to maximize the level of parallel data processing for queries with a heavy workload (e.g., multiple HBase servers can serve a query simultaneously based on their local table regions).

Another complexity we faced in constructing our prototype is that a SQL database has a single point of data consistency, while the HBase database replicates data across multiple nodes. HBase is built upon the Hadoop File system (HDFS), in which a write is not committed until it is guaranteed to be stored by multiple nodes. There are parameters that control how many replications are required before a write returns. For our prototype implementation, we chose a replication factor of 0 (i.e., no replication), but for ongoing investigation in real environments, we would have to use replication to increase availability.

3.7 Discussion

Generality. While Tide is designed and implemented based on vSphere, a centralized datacenter management system, we speculate that techniques used to achieve self-scaling in Tide may also be useful for distributed management systems such as Eucalyptus[85]. Distributed management systems usually assign a subset of hosts to one management node and scale out by adding more management nodes. Since workload bursts are common in virtualized datacenters, they may cause certain management nodes to become "hotspots". In this case, Tide can be deployed on each management node and share a single resource pool for provisioning virtual instances.

Consistency. Management task execution often requires consistency and synchronization. For instance, multiple virtual network reconfiguration tasks could modify the configuration of a shared virtual network at the same time. As a result, vSphere provides several types of lock services to support parallel execution of tasks on one management instance. Tide extends the lock service to support tasks execution over distributed management instances. Specifically, the primary instance runs the lock manager and all instances acquire and release locks by communicating with the primary instance via network. However, the downside is that lock operations have longer latencies compared with local lock operations in vSphere.

Fault tolerance. Tide employs a simple failure model. First, we consider the failure of the primary management instance as the failure of the entire system. Second, the fail-stop failure of one or more virtual management instances is tolerable. Here the term tolerable means the consequence of such a failure could at most cause small degradation of performance.

Tide detects the failure of a virtual instance through a simple heartbeat mechanism. Periodically, a virtual instance sends a heartbeat message to the primary instance to indicate it is alive. The primary instance considers a virtual instance as failed when it misses several heartbeat messages. In case of a virtual instance failure, the primary instance starts a new virtual instance, and assigns all tasks of the failed instance to the new one. The new instance then connects to the corresponding hosts to execute the assigned tasks. Each host records the identity of tasks executed in the past 24 hours. It executes a requested task only if it has not seen the task identity before. This is to avoid re-executing tasks finished by the failed instance.

3.8 Experimental Evaluation

We perform extensive experiments to evaluate the effectiveness of Tide based on both real world workload traces and synthetic workloads. We highlight the key results from our experiments in the following:

- Tide can quickly scale to meet the demand of various workload bursts. In addition, our adaptive provisioning scheme has much lower resource demand compared to fixed-step-provisioning.
- The workload dispatching algorithm of Tide can effectively drive up instance utilization. Its performance is significantly better than eager dispatching and reasonably close to that of the optimal dispatching.

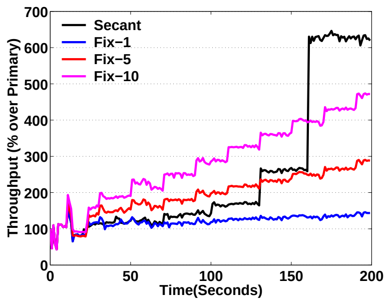
3.8.1 Experiment Setup

Our setup includes a primary vSphere server, a total of 50 virtual vSphere servers, 300 hosts with a total of 3000 virtual machines running on them. The primary vSphere server runs on a Dell PowerEdge 1850 with four 2.8GHz Xeon CPUs and 3.5GB of RAM. All virtual machines installed with vSphere servers are deployed on a set of hosts. We installed vSphere 4.0 on all management instances.

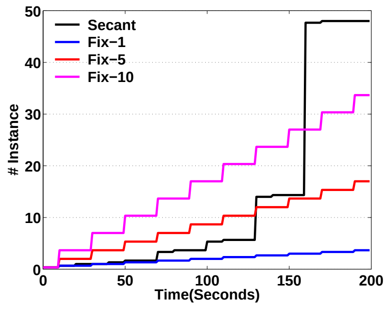
We conduct experiments based on both real world and synthetic workloads. The real world workload is generated from traces we collected from several customer datacenters. The trace data includes management tasks performed over thousands of virtual machines in a three-year period. From this trace data, we generate two types of real world workloads. The first models short-term workload bursts. We use this workload to evaluate the effectiveness of Tide's self-scaling feature. The second models long-term management workload, which contains both regular and bursty workloads. We use this set of workloads to assess the long-term performance of Tide and its resource efficiency. Note that the real world workload trace we use represents the workload of a cloud hosting virtualized datacenters of multiple enterprise users. This workload is heavier than that of an individual enterprise datacenter in general. We also use synthetic workload to measure different aspects of our algorithm, e.g. it allows us to create workload bursts with different characteristics and to measure the impact.

3.8.2 Results

Instance Provisioning. Figure 17 shows the throughput and the number of instances traces of Tide during a self-scaling process when using different provisioning schemes. Here *Secant* refers to the speedup guided provisioning scheme we use in Tide and *Fix-N* is the simple provisioning scheme that adds N instances to the system if it observes throughput improvement in the previous iteration. The workload input for this figure is a period of management workload bursts that lasts 200 seconds and we normalize all throughput by the throughput of the primary instance. The speedup guided scheme has a better provisioning speed compared with all fixed-step schemes. It adds a small number of instances at first and gradually adds more instances in each iteration as it predicts the speedup change rate better. Although Fix-10 performs reasonably well, as witnessed by our later results, it cause significant resource consumption in the long run due to small and frequent workload bursts.



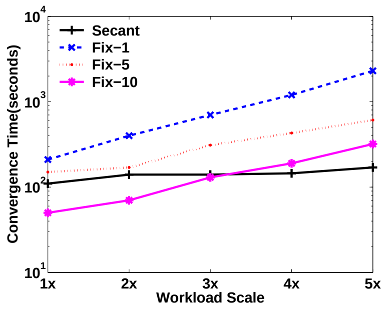
(a)



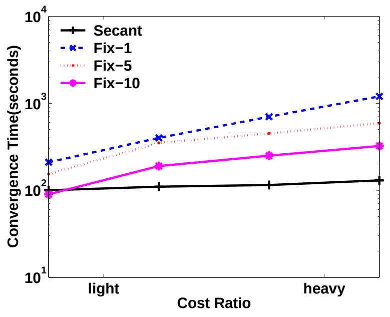
(b)

Figure 17: Performance Trace During Self-Scaling: (a) Throughput Trace; (b) Instance Number Trace

Figure 18 illustrates the convergence time of different provisioning schemes under different workload characteristics. The convergence time measures the time a scheme takes to provision the desirable number of instances, i.e. reaching the stable state where adding more instances improves little throughput. We use synthetic workload to control the workload characteristics. In Figure 18(a) we push the incoming rate of tasks from a base level of 1x to 5x(5 times higher). It is clear that the speedup guided scheme consistently uses less time to converge and its convergence time is barely effected by workload changes. The convergence time of fixed-step schemes such as *FIX-10*, while smaller than that of the speedup guided scheme under small workloads, degrades with increasing workloads. This workload insensitive feature of our speedup guided scheme is particularly appealing as management workloads may vary from time to time. In Figure 18(b), we evaluate different schemes by increasing the workload weight. We rank different types of tasks by their CPU consumption at the management instance. The heavier a workload, the more CPU-intensive tasks it has. Similarly, the speedup guided scheme outperforms fix-step schemes and is insensitive to workload changes.



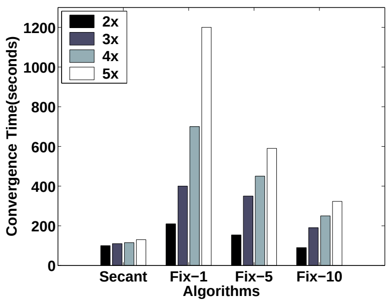
(a)



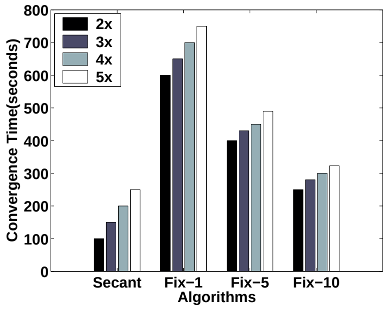
(b)

Figure 18: Convergence Time under Different Workload Characteristics: (a) Increasing Workload Scale; (b) Increasing Workload Weight

In Figure 19, we study the convergence time of different provisioning schemes under different types of workload bursts. Again, we use a synthetic workload as it allows us to create workload bursts of different types. Figure 19(a) shows the convergence time of different schemes under workload bursts whose task incoming rate increases from the base level(1x) to a higher level(2x-5x). We can see that the speedup-guided approach consistently achieves much shorter convergence time compared with other approaches. We



(a) Growing Bursts



(b) Declining Bursts

Figure 19: Convergence Time under Different Types of Bursts

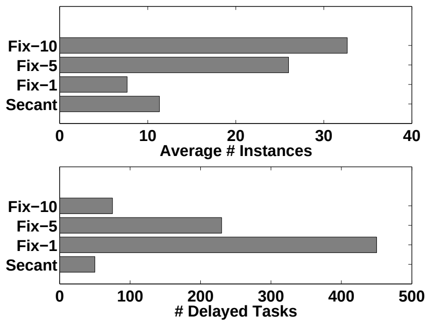


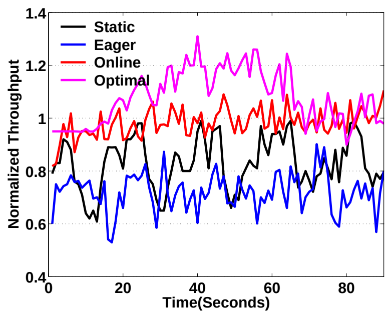
Figure 20: Overall Performance/Efficiency

can observe similar results in Figure 19(b) where the workload bursts drop from a higher level(2x-5x) to the base level(1x). The speedup guided approach has higher convergence time in declining bursts as it relies on fixed-step instance reduction to handle overshooting (Restriction 2). Nevertheless, reducing instances from the steady state does not cause task execution latency.

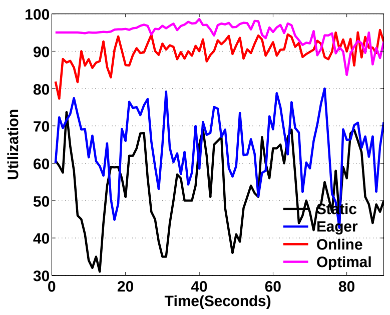
Figure 20 shows the performance and efficiency of different schemes for long-term workloads. We use real world workload traces for this experiment. The workload trace is a combination of management traces from different datacenters and lasts about 1 hour. Here we measure the performance with the accumulated number of tasks that are delayed more than 50% of their normal execution time due to insufficient management capacity. For efficiency, we measure the average number of instances used by different schemes during self-scaling. Clearly, we can see that our scheme causes much smaller number of delayed tasks and yet uses a relatively small number of instances. Compared with our approach, fixed-step schemes either causes too many delayed tasks, e.g., Fix-1, or uses too many instances, e.g., Fix-10. The long-term performance and efficiency results suggest that the speedup-guided approach is cost-effective.

Workload Dispatching. Figure 21 shows the performance trace of different workload dispatching schemes based on a real world workload trace that lasts 100 seconds. We compare four different workload dispatching schemes. The first, *Static*, is the static workload dispatching scheme which assigns hosts to instances in round-robin and never changes the assignment during the execution. The second scheme, *Eager*, is the eager dispatching scheme we discussed in Section 3.5. The eager scheme aggressively reassigns hosts with waiting tasks to another instance with idle work threads and sufficient space. It repeats the process until no such instance exists. The third scheme, *Online*, is the online dispatching scheme we propose. It is the same to the eager scheme except that it reassigns a host only when the workload density ρ of the host is sufficiently large. Finally, the fourth scheme, *Optimal*, is essentially the eager scheme with future workload knowledge. It performs in the same way as the eager scheme. However, it knows not only the current workload of a host but also the future workload on a host. In our experiments, we feed the optimal scheme with workload information from the current moment to 10 seconds in the future. Note that the optimal scheme is not available in the real world. We use the optimal scheme to evaluate how close the online scheme can reach the ideal performance.

Figure 21(a) suggests that the online scheme has a similar throughput to the optimal scheme. Note that the optimal trace sometimes reaches a normalized throughput larger



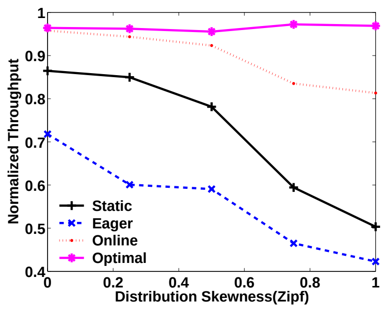
(a) Throughput



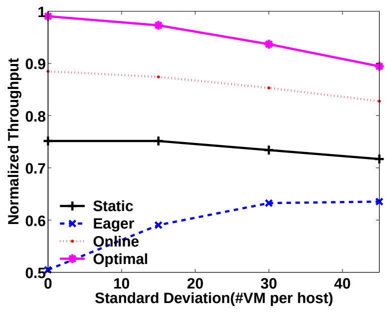
(b) Utilization

Figure 21: Performance Trace of Different Workload Dispatching Schemes

than 1 because we use the average incoming task rate to normalize throughput and the actual throughput may be higher than the average. The throughput of the static and the eager schemes, however, is not only lower but also varies heavily. The static scheme does not adjust host assignment, and thus, suffers from performance degradation when workload shifts from one set of hosts to another set of hosts. The eager scheme, on the other hand, aggressively reassigns hosts based on only current workload information and causes low available space on all instances. As a result, when a host receives high workload later, the eager scheme may miss the chance of finding a lightly loaded instance with sufficient space to manage the host. In fact, the eager scheme performs even worse than the static scheme due to reconnection latencies caused by host reassignment. Figure 21(b) shows the average instance utilization of different schemes. It suggests that our online scheme has similar performance as the optimal scheme, while the static and eager schemes result in much lower instance utilization.



(a) Varying Workload

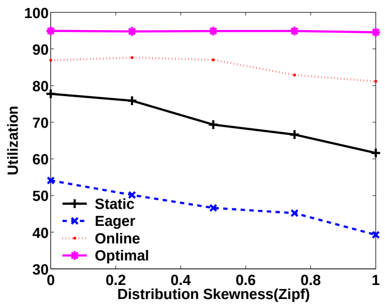


(b) Varying Host Footprints

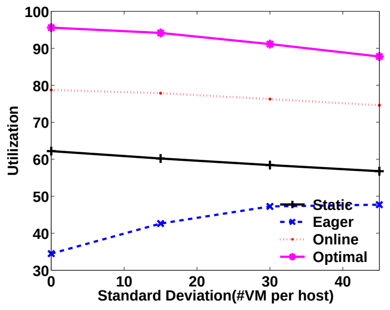
Figure 22: Throughput under Different Workload and Infrastructure

We next evaluate the performance of these four schemes given different workloads (synthetic) and infrastructure. In Figure 22(a), we vary the skewness in the workload distribution and examine the impact on workload dispatching schemes. Specifically, we distribute tasks to hosts based on Zipf distribution and vary the skewness s from 0 (equivalent to uniform distribution) to 1. We can see that the throughput of the static and the eager scheme degrades heavily as the skewness increases, while the online scheme consistently outperforms the static and eager schemes. This is because reservation made by the online scheme leaves space for hosts with high workload at a later time, while the static scheme and the eager scheme miss such opportunities.

In Figure 22(b), we vary the footprint size distribution between hosts. The footprint size here refers to the number of virtual machines on a host. In our experiment, we vary the standard deviation of footprint sizes from 0 to 45 (with a mean equals to 55). Clearly, the online scheme has steady performance advantages over the static and eager schemes. Accordingly, we find that the utilization of the online dispatching scheme is also consistently



(a) Varying Workload



(b) Varying Host Footprints

Figure 23: Utilization under Different Workload and Infrastructure

higher than that of the static and eager schemes in Figure 23. Note that the performance of the online scheme is also reasonably close to that of the optimal scheme.

3.9 Related Work

There are a number of management systems for virtualized environments. Usher[75] is a modular open-source virtual machine management framework from UCSD. Virtual Workspaces[59] is a Globus-based[43] system for provisioning workspaces (i.e., VMs), which leverages several pre-existing solutions developed in the grid computing arena. The Cluster-on-demand[28] project focuses on the provisioning of virtual machines for scientific computing applications. oVirt[7] is a Web-based virtual machine management console. Several commercial virtualization management products are also available, including vSphere[116] from VMware and System Center[6] from Microsoft. Enomalism[5] is a commercial open-source cloud software infrastructure. Despite the large number of existing systems, few works have studied their performance.

The concept of auto-scaling is not new, but we extend it beyond application-level scaling and apply it to the management workload. Application-level auto-scaling[13] dynamically adjusts the number of server instances running an application according to application usage. In fact, providing application-level auto-scaling to many applications may further increase the intensity and burstiness of the management workload. For example, creating many web server VMs to handle a flash crowd of HTTP requests requires fast execution of a large number of management operations (clones and power ons). Therefore, we speculate that Tide may provide better support for application auto-scaling compared to management systems with fixed capacity. The problems in application-level auto-scaling are also quite different from those we study. E.g., application workload dispatching is often request-based and can be implemented with off-the-shelf techniques, while the workload dispatching in Tide is host-based. Everest[84] is a transparent layer that allows data written to an overloaded volume to be temporarily off-loaded into a short-term virtual store. The storage performance scaling problem studied in this chapter is quite different from the one we study in Tide. Recent works [118, 66] on flexible monitoring/management infrastructure study designs that allow users to deploy different monitoring/analysis approaches under the same framework. In contrast, we propose Tide to enable automatic scaling of the management/monitoring service.

Tide uses multiple small-footprint server instances to boost system throughput via parallel task execution. We choose to use small-footprint server instances because they enable fast instantiation and minimum latency in each provisioning cycle. FAWN[16] is a cluster architecture based on small footprint nodes with embedded CPUs and local flash storage. It balances computation and I/O capabilities to enable efficient, massively parallel access to data. However, the focus of FAWN is on energy-efficient computing.

The execution of management tasks in a virtualized datacenter involves both the management system and hosts running virtual machines. SnowFlock[67] studies rapid group-instantiation of virtual machines on virtualized host side. This work is complementary to ours as we focus on efficient task execution in the management system.

Several researchers have studied the problem of resource provisioning for meeting application-level quality of service (QoS). Padala[89], et al proposed an resource control scheme that dynamically adjusts the resource shares to individual tiers to meet QoS goals. Their approach is based on control theory and requires performance modeling. Doyle and et al.[41] studied a model-based resource provisioning approach for web services in shared server clusters based on queuing theory. Compared with these approaches, the speedup-guided approach in Tide does not require system-dependent performance modeling and may adapt to different environments easier. In addition, we also study maximizing instance utilization through distributed workload dispatching. In addition, while these approaches do not seek to minimize provisioning latency, we try to reduce provisioning latency at both the algorithmic level and the implementation level as it is essential to self-scaling.

The workload dispatching problem in Tide is quite different from traditional workload dispatching in distributed systems such as MapReduce[36] where the assignment is per-task. Instead, the assignment unit in Tide is a host, each of which observes a stream of incoming management tasks. This fundamental difference makes static assignment less useful as each host may observe a different workload over time. Our workload dispatching algorithm is based on the online algorithm proposed by Zhou[128], et al. We adapted their algorithm for our setting where the dispatching decisions are made continuously.

CHAPTER IV

STATE MONITORING IN CLOUD DATACENTERS

4.1 Introduction

Cloud datacenters represent the new generation of datacenters that promote on-demand provisioning of computing resources and services. Amazon's Elastic Computer Cloud(EC2)[14] is an example of such cloud datacenters. A typical Cloud application in such Cloud datacenters may spread over a large number of computing nodes. Serving Cloud applications over multiple networked nodes also provides other attractive features, such as flexibility, reliability and cost-effectiveness. Thus, state monitoring becomes an indispensable capability for achieving on-demand resource provisioning in Cloud datacenters. However, the scale of Cloud datacenters and the diversity of application specific metrics pose significant challenges on both system and data aspects of datacenter monitoring for a number of reasons.

First, the tremendous amount of events, limited resources and system failures often raise a number of system-level issues in datacenter monitoring:

- *Event Capturing.* Applications, OS, servers, network devices can generate formidable amount of events, which makes directly storing and searching these events infeasible. To address this issue, Bhatia et al. [21] proposed Chopstix, a tool that uses approximate data collection techniques to efficiently collect a rich set of system-wide data in large-scale production systems.

- *Resource Consumption.* Servers usually have limited resources available for monitoring. Assigning monitoring tasks and organizing monitoring overlays without considering this fact may lead to unreliable monitoring results. Jain et al.[54] proposed a self-tuning monitoring overlay to trade precision and workload. Meng et al. [79] studied the problem of monitoring network construction for multiple monitoring tasks without overloading member hosts.

- *Reliability.* Failures of server, network links can lead to inconsistent monitoring results. Jain et al. [55] introduced and implemented a new consistency metric for large-scale monitoring. The new metric indicates the precision of monitoring results, and thus, can identify inconsistent results caused by system failures.

Second, large-scale monitoring often involves processing large amount of monitoring data in a distributed manner. Such computing paradigm also introduces several challenges at the data management level:

- *Distributed Aggregation.* The ability of summarizing information from voluminous distributed monitored values is critical for datacenter monitoring. Previous work proposed several efficient algorithms for different aggregation over distributed stream values. Babcock et al. [19] studied the problem of monitoring top-k items over physically distributed streams. Olston et al. [86] introduced an efficient algorithm for computing sums and counts of items over distributed streams. As its distinct feature, the proposed algorithm can achieve efficiency by trading precision for communication overhead. Cormode et al.[34] proposed an approach for approximate quantile summaries with provable approximation guarantees over distributed streams.

- *Shared Aggregation.* Different monitoring tasks may share some similarities. Running similar tasks in an isolated manner may lead to unnecessary resource consumption. Krishnamurthy et al. [64] developed techniques for binding commonalities among monitoring queries and sharing work between them.

In this chapter, we study state monitoring at Cloud datacenters, which can be viewed as a Cloud state management issue, as it mainly involves collecting local state information and evaluating aggregated distributed values against pre-defined monitoring criteria. A key challenge for efficient state monitoring is meeting the two demanding objectives: high level of correctness, which ensures zero or very low error rate, and high communication efficiency, which requires minimal communication cost in detecting critical state violation.

4.1.1 State Monitoring

Despite the distributed nature of Cloud-hosted applications, application owners often need to monitor the global state of deployed applications for various purposes. For instance, Amazon's CloudWatch [1] enables users to monitor the overall request rate on a web application deployed over multiple server instances. Users can receive a state alert when the overall request rate exceeds a threshold, e.g. the capacity limit of provisioned server instances. In this case, users can deploy the web application on more server instances to increase throughput.

As another example, service providers who offer software-as-a-service to organizations often need to perform distributed rate limiting (DRL) to restrict each organization to use the software within its purchased level (e.g. 100 simultaneous sessions). Because software services are usually deployed over distributed servers in one or multiple datacenters, they require DRL to check if the total number of running sessions from one organization at all servers is within a certain threshold.

We refer to this type of monitoring as *state monitoring*, which continuously evaluates if a certain aspect of the distributed application, e.g. the overall request rate, deviates from a normal state. State monitoring is widely used in many applications. Examples also include:

EXAMPLE 1. *Traffic Engineering: monitoring the overall traffic from an organization's sub-network (consists of distributed hosts) to the Internet.*

EXAMPLE 2. *Quality of Service: monitoring and adjusting the total delay of a flow which is the sum of the actual delay in each router on its path.*

EXAMPLE 3. *Fighting DoS Attack: detecting DoS attack by counting SYN packets* *arriving at different hosts within a sub-network.*

EXAMPLE 4. *Botnet Detection: tracking the overall simultaneous TCP connections from a set of hosts to a given destination.*

State monitoring in datacenters poses two fundamental requirements. First, given the serious outcome of incorrect monitoring results, state monitoring must deliver correct monitoring results[24]. A false state alert in the previous CloudWatch example would cause provisioning of new server instances which is clearly unnecessary and expensive. Missing a state alert is even worse as the application gets overloaded without new server instances, which eventually causes potential customers to give up the application due to poor performance. This correctness requirement still holds even if monitored values contain momentary bursts and outliers.

Second, communication related to state monitoring should be as little as possible[70, 73, 69]. Datacenters usually run a large number of state monitoring tasks for application and infrastructure management[14]. As monitoring communication consumes both bandwidth and considerable CPU cycles[79], state monitoring should minimize communication. This is especially important for infrastructure services such as EC2, as computing resources directly generate revenues.

One intuitive state monitoring approach is the instantaneous state monitoring, which triggers a state alert whenever a predefined threshold is violated. This approach, though makes algorithm design easy, idealizes real world monitoring scenarios. As unpredictable short-term bursts in monitored values are very common for Internet applications[37, 49, 99], instantaneous state monitoring may cause frequent and unnecessary state alerts. In the previous example, momentary HTTP request bursts trigger unnecessary state alerts whenever their rates exceed the threshold. Furthermore, since state alerts usually invoke expensive counter-measures, e.g. allocating and deploying new web server instances, unnecessary state alerts may cause significant resource loss. Surprisingly, we find most of the existing work to date[39, 86, 60, 97, 11, 57] deals only with this type of state monitoring.

4.1.2 Overview of Our Approach

In this chapter, we introduce the concept of window-based state monitoring and devise a distributed *WIndow-based StatE monitoring* (WISE) framework for Cloud datacenters. Window-based state monitoring triggers state alerts only when observing *continuous* state violation within a specified time window. It is developed based on the widely recognized observation that state violation within a short period may simply indicate the dynamics of the runtime system and it does not necessarily trigger a global state violation. Thus, with the persistence checking window, window-based state monitoring gains immunity to momentary monitoring value bursts and unpredictable outliers.

In addition to filtering unnecessary alerts, window-based state monitoring explores monitoring time windows at distributed nodes to yield significant communication savings. Although the window-based state monitoring approach was first introduced in [83], the focus of our earlier results was mainly on the basic approach to window-based state monitoring with centralized parameter tuning to demonstrate and evaluate its advantage in monitoring cost saving compared to instantaneous state monitoring. In this chapter, we identify that this basic approach to window-based state monitoring may not scale well in the presence of lager number of monitoring nodes. We present an improved window based monitoring approach that improves our basic approach along several dimensions. First, we present the architectural design of the WISE system and its deployment options (Section 4.3.1). Second, to address the scalability issue of the basic WISE, we develop a distributed parameter tuning scheme to support large scale distributed monitoring (Section 4.5.4). This distributed scheme enables each monitoring node to search and tune its monitoring parameters in a reactive manner based on its observations of state update events occurred, without requiring global information. It enables WISE to scale to a much larger number of nodes compared with the centralized scheme. Third, we design two concrete optimization techniques, aiming at minimizing the communication cost between a coordinator and its monitoring nodes. The first optimization is dedicated to enhance the effectiveness of the global pull procedure at the coordinator by reducing the communication cost for global pulls, while ensuring the correctness of the monitoring algorithm. The second optimization aims at reducing unnecessary global polls by reporting more information of local violations at monitoring nodes (Section 4.6). Finally, we have conducted extensive empirical studies on the scalability of the distributed parameter tuning scheme compared to the centralized scheme appeared first in [83], and evaluated the effectiveness of both the distributed WISE solution and the two optimization techniques, compared to the basic WISE approach (Section 4.7.2).

In summary, this chapter makes three unique contributions. First, WISE employs a novel distributed state monitoring algorithm that deploys time windows for message filtering and achieves communication efficiency by intelligently avoiding collecting global information. More importantly, it also guarantees monitoring correctness. Second, WISE uses a distributed parameter tuning scheme to tune local monitoring parameters at each distributed node and uses a sophisticated cost model to carefully choose parameters that can minimize the communication cost. As a result, this scheme scales much better than the centralized scheme presented in [83]. Last but not the least, we develop a set of optimization techniques to optimize the performance of the fully distributed WISE. Compared with other works using statistical techniques to avoid false positive results in runtime events monitoring such as performance anomaly detection [119, 117], we focus on developing efficient distributed monitoring algorithms for simple, widely used filtering techniques such as window based monitoring.

We conducted extensive experiments over both real world and synthetic monitoring traces, and show that WISE incurs a communication reduction from 50% to 90% compared with existing instantaneous monitoring approaches and simple alternative window based schemes. We also compare the original WISE with the improved WISE on various aspects.

Our results suggest that the improved WISE is more desirable for large-scale datacenter monitoring.

4.1.3 Outline

The rest of this chapter is organized as follows. Section 4.2 introduces the preliminaries and defines the problem of window based state monitoring. Section 4.3 gives an overview of our approach. Section 4.4 presents the detail of the WISE monitoring algorithm. Section 4.5 describes our scalable parameter setting scheme. We discuss optimization techniques to further improve the performance of WISE in Section 4.6. Section 4.7 presents the experimental evaluation. Section 6.5 discusses the related work.

4.2 Preliminaries

We consider a state monitoring task involving a set N of nodes where $|N| = n$. Among these n nodes, one is selected to be a coordinator which performs global operations such as collecting monitored values from other nodes and triggering state alerts. For a given monitoring task, node i locally observes a variable v_i which is continuously updated at each time unit. The value of v_i at time unit t is $v_i(t)$ and we assume $v_i(t)$ is correctly observed. When necessary, each monitor node can communicate with the coordinator by sending or receiving messages. We consider that communication is reliable and its delay is negligible in the context of datacenter state monitoring. As communication cost is of concern, we are interested in the total number of messages caused by monitoring. We also consider the size of messages in our experiment.

A state monitoring task continuously evaluates a certain monitored state is normal or abnormal. Similar to previous work[39, 86, 60, 97, 11, 57], we distinguish states based on sum aggregate of monitored values. For instance, we determine whether a web application is overloaded based on the sum of HTTP request rates at different hosts. We use sum aggregates because they are widely applied and also simplify our discussion, although our approach supports any aggregate that linearly combines values from nodes.

4.2.1 The Instantaneous State Monitoring

The instantaneous state monitoring model[39, 86, 60, 97, 11, 57] detects state alerts by comparing the current aggregate value with a global threshold. Specifically, given $v_i(t), i \in [1, n]$ and the global threshold T, it considers the state at time t to be abnormal and triggers a state alert if $\sum_{i=1}^{n} v_i(t) > T$, which we refer to as *global violation*.

To perform instantaneous state monitoring, the line of existing work decomposes the global threshold T into a set of local thresholds T_i for each monitor node i such that $\sum_{i=1}^{n} T_i \leqslant T$. As a result, as long as $v_i(t) \leqslant T_i, \forall i \in [1, n]$, i.e. the monitored value at any node is lower or equal to its local threshold, the global threshold is satisfied because $\sum_{i=1}^{n} v_i(t) \leqslant \sum_{i=1}^{n} T_i \leqslant T$. Clearly, no communication is necessary in this case. When $v_i(t) > T_i$ on node i, it is possible that $\sum_{i=1}^{n} v_i(t) > T$ (global violation). In this case, node i sends a message to the coordinator to report *local violation* with the value $v_i(t)$. The coordinator, after receiving the local violation report, invokes a *global poll* procedure where it notifies other nodes to report their local values, and then determines whether $\sum_{i=1}^{n} v_i(t) \leqslant T$. The focus of existing work is to find optimal local threshold values that minimize the overall communication cost.

4.2.2 The Window-based State Monitoring

As monitored values often contain momentary bursts and outliers, instantaneous state monitoring [49] is subject to cause frequent and unnecessary state alerts, which could further lead to unnecessary counter-measures. Since short periods of state violation are often well acceptable, a more practical monitoring model should tolerate momentary state violation and capture only continuous one. Therefore, we introduce window-based state monitoring which triggers state alerts only when the normal state is *continuously* violated for L time units.

We study window-based state monitoring instead of other possible forms of state monitoring for two reasons. First, we believe continuous violation is the fundamental sign of

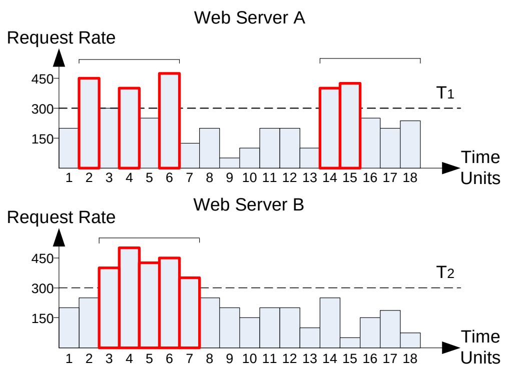


Figure 24: A Motivating Example

established abnormality. Second, window-based state monitoring tasks are easy to configure, because the window size L is essentially the tolerable time of abnormal state, e.g. degraded service quality, which is known to service providers.

4.2.3 Problem Definition

Our study focuses on finding efficient ways to perform distributed window-based state monitoring, as this problem is difficult to solve and, to the best of our knowledge, has not been addressed before. Formally, we define the distributed window-based state monitoring problem as follows:

Problem Statement 3 *Given the threshold T, the size L of the monitoring window, and n monitor nodes with values $v_i(t), i \in [1, n]$ at time t, devise an algorithm that triggers state alerts only when $\sum_{i=1}^{n} v_i(t-j) > T, \forall j \in [0, L-1]$ at any t while minimizing the associated communication cost.*

Solving this problem, however, is challenging, as it requires careful handling of monitoring windows at distributed nodes to ensure both communication efficiency and monitoring correctness. Simple solutions such as applying modified instantaneous monitoring approaches either fail to minimize communication or miss state alerts. We next present a motivating example to show the reason as well as some insights into the solution.

Figure 56 shows a snippet of HTTP request rate traces collected from two web servers in a geographically distributed server farm[18], where time is slotted into 5-second units. Let us first consider an instantaneous monitoring task which triggers state alerts when the sum of request rates at two servers exceeds $T = 600$. For simplicity, we assume server A and B have the same local thresholds $T_1 = T_2 = T/2 = 300$, as indicated by dashed lines. A local violation happens when a bar raises above a dashed line, as indicated by bars with red borders.

In the example, server A and B report local violation respectively at time unit 2,4,6,14,15, and time unit 3-7, which generates 10 messages. When receives local violation reports, the coordinator invokes global polls at time unit 2,3,5,7,14,15 to collect values from the server that did not report local violation. No global poll is necessary at time unit 4 and 6 as the coordinator knows local values of both servers from their local violation reports. Each global poll includes one message for notification and one message for sending back a local value, and all global polls generate $6 \times 2 = 12$ messages. Thus, the total message number is $10 + 12 = 22$.

4.2.3.1 Applying Instantaneous Monitoring

Now we perform window-based state monitoring to determine whether there exists continuous global violation against T lasting for $L = 8$ time units. We start with the most intuitive approach, –applying the instantaneous monitoring algorithm. Specifically, a monitor node i still evaluates whether $v_i(t) > T_i$ and reports local violation to the coordinator if it is true. The coordinator then invokes a global poll to determine if $\sum v_i(t) > T$. The only difference is that the coordinator triggers state alerts only when observing continuous global violation of 8 time units. As a result, the communication cost is the same as before, 22 messages. Note that 22 messages are generated for only 2 monitor nodes and all messages have to be processed by the coordinator. Our experiment suggests that the total message number in this scheme grows quickly with increasing monitor nodes. This can cause significant bandwidth and CPU cycle consumption at the coordinator, which limits the scalability of monitoring.

4.2.3.2 Saving Communication at The Coordinator

In fact, invoking a global poll for every local violation is not necessary. Since state alerts require continuous global violation, the coordinator can delay global polls unless it observes 8 continuous time units with local violation. When it observes a time unit t with no local violation, it can clear all pending global polls, as the violation is not continuous, and thus, avoids unnecessary communication. This modified scheme avoids all 6 global polls, as no local violation exists at time unit 8 and 16. Therefore, by avoiding unnecessary communication at the coordinator side, the total message number is reduced to 10.

4.2.3.3 Saving Communication at Monitor Nodes

Reducing communication at monitor nodes is relatively more difficult. One may propose to let each node report the beginning and the end of a continuous local violation period, instead of reporting for each time unit with local violation. This scheme, which we refer to as *double-reporting*, saves 3 messages on server B by reporting at time 3 and 7, but performs poorly (8 messages) on server A as each violation period costs two messages, even when it is short (e.g. time unit 2, 4, 6). The total message number is still 10. One may also suggest monitor nodes to report only the end of violation period for less communication. This *end-reporting* scheme, however, fails to ensure monitoring correctness. Assume server A observes local violation throughout time unit 2-10 and $\sum v_i(t) > T, \forall t \in [2, 10]$. The coordinator inevitably fails to trigger a state alert at time unit 9 without knowing that server A has started to observe local violation at time unit 2.

4.2.3.4 Insights and Challenges

One solution is to lower the granularity of local violation reporting, as approximate information on local violation is often adequate to rule out state alerts. Monitor nodes, after reporting one local violation, can employ message filtering time windows with pre-defined lengths to suppress subsequent local violation reporting messages. For instance, assume both server A and B use 5-time-unit filtering windows. Server A reports local violation at time unit 2, and then enters a filtering window, during which it avoids to report at time unit 4 and 6. Similarly, it reports at time 14 and server B reports once at time 3. At the coordinator side, as filtering windows span 5 time units, the worst case that one reported local violation could imply is a local violation period of 5 time units. Thus, the worst case scenario indicated by the three reports is global violation in time units 2-7 and 14-18, which suggests no state alert exists. The resulting message number is 3, a 86.36% communication reduction over 22 messages.

While the above approach seems promising, devising a complete solution requires answers to several fundamental questions. Example questions include how to process reported local violation and filtering windows at the coordinator side to guarantee monitoring correctness? how to tune monitoring parameters, e.g. local threshold and filtering window size, at each node to achieve minimum communication? and how to optimize different subroutines (e.g. global poll) to further reduce communication cost? In addition, datacenter monitoring often requires many tasks, and each task could potentially involve hundreds, even thousands, of monitor nodes. Thus, it is also important to address questions such as what architecture should WISE employ to support such deployment, and how to achieve high scalability for tasks with many monitor nodes? In the subsequent sections, we present the design and development of WISE, a system that performs accurate and efficient window-based state monitoring over a network of distributed monitor nodes.

4.3 WISE Monitoring System

We present an overview of the WISE monitoring system in this section. We first introduce the architecture and deployment of WISE, and then, describe important components of WISE.

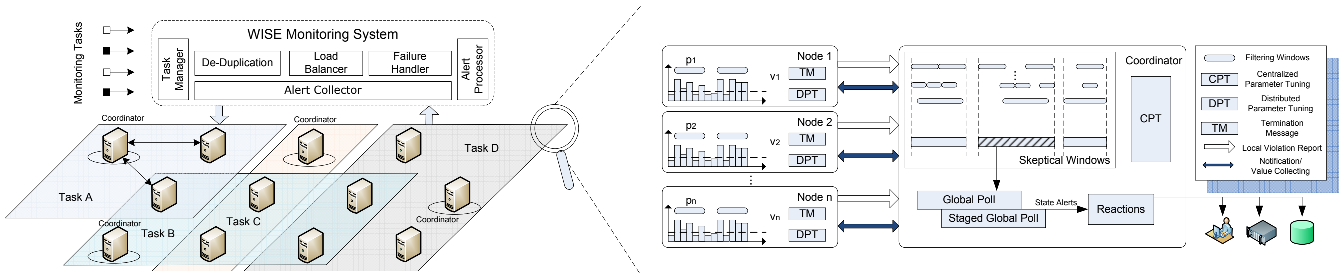


Figure 25: WISE Monitoring System

4.3.1 Architecture and Deployment

The WISE monitoring system takes the description of window-based monitoring tasks as input, continuously watches the state changes over the nodes being monitored, and triggers alerts when the state change meets the specified threshold. The description of a window-based monitoring task specifies the following five conditions: (i) the metric to be monitored at a node (e.g. incoming HTTP request rates), (ii) the set of nodes associated with the monitoring task (N), (iii) the global value threshold (T), (iv) the monitoring time window (L) and (v) the counter-measures to take when a state alert is triggered. The left side of Figure 25 illustrates a sketch of the architectural design of the WISE system and a deployment example of monitoring tasks. Given a set of monitoring tasks, the system first scans for identical monitoring tasks and removes duplicated ones. It then deploys monitoring tasks on their associated nodes. During the monitoring process, the system collects reported state alerts from deployed monitoring tasks and processes these alerts according to specified counter-measures. It also watches machine failures that may impact deployed monitoring tasks. For instance, if one machine becomes unavailable, it identifies monitoring tasks involved with the machine and marks the corresponding monitoring results as unreliable to prevent false positive or negative results.

The deployment example in Figure 25 shows four monitoring tasks running over 12 hosts. One host may be involved with multiple monitoring tasks. The deployment may involve load balancing and monitoring network construction[79]. For example, the system may choose hosts involved with few monitoring tasks to be coordinators as coordinators consume more CPU and bandwidth resources compared with monitor nodes. In the rest of this chapter, we focus on developing efficient schemes for a single monitoring task. We leave other problems such as multi-task optimization as our future work.

4.3.2 WISE Monitoring Approach

We now focus on the three technical developments that form the core of the WISE monitoring approach: the WISE monitoring algorithm, the monitoring parameter tuning schemes and performance optimization techniques. The right side of Figure 25 shows a high level view of the WISE monitoring approach.

4.3.2.1 The Monitoring Algorithm

The idea behind the WISE monitoring algorithm is to report partial information on local violation series at the monitor node side to save communication cost. The coordinator then uses such partial information to determine whether it is possible to detect state alerts. The coordinator collects further information only when the possibility of detecting state alerts cannot be ruled out.

Specifically, the monitor node side algorithm employs two monitoring parameters, the local threshold T_i and the filtering window size p_i. When detects local violation($v_i(t) > T_i$), a monitor node i sends a local violation report and starts a filtering window with size p_i during which it only records monitored values and does not send violation reports.

The coordinator considers a reported local violation at node i as possible continuous local violation spanning p_i time units, since it does not know the complete violation information within the corresponding filtering window. It then "merges" possible continuous local violation reported from different nodes into a potential global continuous violation against T, namely *skeptical window*. The skeptical window holds a nature invariant that no state alert is necessary as long as the length of the skeptical window does not exceed L. The coordinator continuously maintains the skeptical window and tries to rule out the possibility of state alerts based on this invariant. It invokes a global poll to collect complete violation information only when the length of the skeptical window exceeds L.

Intuition. The WISE monitoring algorithm makes two effects to achieve communication efficiency. One is to avoid unnecessary global polls by optimistically delaying global polls, because later observed time units with no local violation indicate that previous global violation is not continuous. The other is to avoid frequent local violation reporting with monitor node side filtering windows. Filtering windows, when their sizes are properly tuned (Section 4.5), can save significant communication from frequently reporting local violation without noticeably diminishing the chance of ruling out state alerts and avoiding global polls. In addition, it ensures monitoring correctness as it always considers the worst case based on received partial information.

4.3.2.2 Scalable Parameter Tuning

State monitoring environments are usually heavily diversified. They may involve monitoring tasks with very different monitoring threshold T and time window L, as well as heterogeneous monitored value distributions across different nodes. As a result, monitoring parameters, i.e. T_i and p_i, should be properly tuned towards the given monitoring task and monitored value patterns for the best communication efficiency. For instance, if a given state monitoring task tries to capture a very rare event, monitor nodes should employ large filtering windows to deliver coarse information to maximally save communication. As another example, if a node often observes higher monitored values compared with other nodes, it should be assigned with relatively higher T_i accordingly.

To provide such flexibility, we proposed a centralized parameter tuning scheme. The centralized tuning scheme runs at the coordinator and setting the parameters for all monitor nodes based on collected information on monitored value distribution. The centralized parameter tuning scheme has one drawback that it requires collecting of global information and performs intensive computation on the coordinator. Given the scale of datacenter monitoring and the exponential increasing nature of search space, the centralized tuning scheme may cause significant resource consumption on the coordinator and fail to find good parameters.

To address this issue, we develop a distributed parameter tuning scheme that avoids centralized information collecting and parameter searching. The distributed scheme runs at each monitor node. Each node tunes its local monitoring parameters based on observed events in a reactive manner. This scheme may produce slightly less efficient parameters compared with those generated by the centralized scheme because it tunes parameters based on local information. Nevertheless, its features such as avoiding searching the entire solution space and limited inter-node communication make it a desirable parameter tuning scheme for large-scale monitoring tasks.

4.3.2.3 Performance Optimization

In addition to improve the basic WISE approach with distributed parameter tuning, we also devise two novel performance optimization techniques, *the staged global poll* and *the termination message*, to further minimize the communication cost between a coordinator node and its monitoring nodes.

The staged global poll optimization divides the original global poll process into several stages. Each stage tries to rule out or confirm state alerts based on a fraction of monitored values that would be collected by the original global poll. Since later stages can be avoided if a previous stage can decide whether a state alert exists, the staged global poll reduces considerable communication. The termination message based optimization deals with “over-reported” local violation periods, which only contain little local violation and may increase the chance of invoking global poll. It tries to remove “over-reported” local violation periods by sending an extra message at the end of a filtering window to indicate real local violation.

In this chapter, we not only provide the algorithmic design but also provide correctness analysis and usage model for both techniques.

4.4 The Monitoring Algorithm

We present the detail of WISE monitoring algorithm in this section. In addition, we also explain why WISE monitoring algorithm guarantees monitoring correctness and theoretically analyze its communication efficiency.

4.4.1 Algorithm Description

WISE monitoring algorithm consists of two parts, the monitor node side algorithm and the coordinator side algorithm:

4.4.1.1 The Monitor Node Side

A monitor node i reports partial information of local violation based on two monitoring parameters, local threshold T_i and filtering window size p_i. Local thresholds of different nodes satisfy $\sum_{i=1}^{n} T_i \leq T$. This restriction ensures the sum of monitored values at all nodes does not exceed T if each value is smaller than its corresponding local threshold.

The filtering window size is the time length of a filtering time window and is defined over $[0, L]$. Specifically, filtering windows are defined as follows.

Definition 5 *A* filtering window *τ of node i is p_i continuous time units during which node i does not send local violation reports even if it observes $v_i(t) > T_i$ where t is a time unit within τ. In addition, we use $|\tau|$ to represent the remaining length of a filtering window τ, $t_s(\tau)$ and $t_e(\tau)$ to denote the start time and the end time of τ. If $p_i = 0$, $t_s(\tau) = t_e(\tau)$ and $|\tau| = 0$.*

When a node i detects $v_i(t) > T_i$ at time unit t and if it is currently not in a filtering window($|\tau| = 0$), it sends a local violation report to the coordinator, and then enters a filtering window by setting $|\tau| = p_i$. During a filtering window($|\tau| > 0$), it does not report local violation and decreases $|\tau|$ by 1 in every time unit. Node i starts to detect and report violation again only after $|\tau| = 0$. For now, we assume T_i and p_i are given for each node. We will introduce techniques for selecting proper values for T_i and p_i later.

4.4.1.2 The Coordinator Side

The coordinator side algorithm "reassembles" potential periods of local violation indicated by local violation reports into a potential period of continuous global violation, which we refer to as the *skeptical window*. The skeptical window essentially measures the length of the most recent continuous global violation in the worst case. The coordinator considers reported local violation from node i as continuous local violation lasting p_i time units, i.e. assuming filtering windows fully filled with local violation. It concatenates reported filtering windows that overlap in time into the skeptical window, which is defined as follows:

Definition 6 *A* skeptical window κ *is a period of time consisting of most recent overlapped filtering windows related with reported local violation since last global poll. Initially, the size of a skeptical window $|\kappa|$ is 0. Given a set of filtering windows $\mathbb{T} = \{\tau_i | i \in [1, n]\}$ observed at time t, κ can be updated as follows:*

$$t_s(\kappa') \leftarrow \begin{cases} t_s(\kappa) & t_e(\kappa) \geq t \\ t & otherwise \end{cases} \tag{3}$$

$$t_e(\kappa') \leftarrow \begin{cases} t + \max\limits_{\forall \tau_i \in \mathbb{T}} \{t_e(\kappa) - t, |\tau_i|\} & t_e(\kappa) \geq t \\ \max\limits_{\forall \tau_i \in \mathbb{T}} \{t, t_e(\tau_i)\} & otherwise \end{cases} \tag{4}$$

where κ' is the updated skeptical window, $t_s(\cdot)$ and $t_e(\cdot)$ is the start and the end time of a window. In addition, $|\kappa| = t_e(\kappa) - t_s(\kappa) + 1$. In our motivating example, server A and B with $p_A = p_B = 5$ report local violation at time 2 and 3 respectively. The corresponding skeptical window covers both filtering windows as they overlap, and thus, spans from time 2 to time 7. Figure 26 shows an illustrative example of skeptical windows.

When $t - t_s(\kappa) = L$, it indicates that there may exist continuous local violation for the last L time units (which could lead to continuous global violation of L time units).

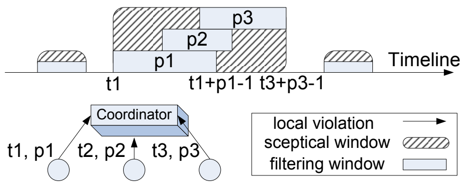


Figure 26: Filtering Windows and Skeptical Windows.

Thus, the coordinator invokes a global poll to determine whether a state alert exists. The coordinator first notifies all nodes about the global poll, and then, each node sends its buffered $v_i(t-j), j \in [0, t']$, where $0 < t' \leqslant L$, to the coordinator in one message. Here t' depends on how many past values are known to the coordinator, as previous global polls and local violation also provides past v_i values. After a global poll, if the coordinator detects continuous global violation of L time units, i.e. $\sum_{i=1}^{n} v_i(t-j) > T, \forall j \in [0, L-1]$, it triggers a state alert and set $|\kappa| = 0$ before continuing. Otherwise, it updates κ according to received v_i. Clearly, the computation cost of both monitor node and coordinator algorithms is trivial.

Filtering windows greatly reduce communication on local violation reporting, but may also cause overestimated local violation periods at the coordinator when filtering windows cover time units with no local violation. This, however, rarely leads to less chance of ruling out global polls and noteworthy increased cost in global polls. First, state alerts are usually rare events. With filtering windows, the coordinator still finds enough "gaps", i.e. time units with no local violation, between reported filtering windows before skeptical window size grows to L. Second, the parameter tuning schemes we introduce later set proper filtering window sizes so that the saving in local violation reporting always exceeds the loss in global polls. Last but not the least, we also develop a staged global poll procedure in Section 4.6 which significantly reduces communication cost in global polls.

4.4.2 Correctness

The WISE monitoring algorithm guarantees monitoring correctness because of two reasons. First, the coordinator never misses state alerts (false negative), as the skeptical window represents the worst case scenario of continuous global violation. Second, the coordinator never triggers false state alerts (false positive) as it triggers state alerts only after examining the complete local violation information. Theorem 6 presents the correctness guarantee of the WISE algorithm.

Theorem 6 *Given a monitoring task (T, L, N), the WISE algorithm triggers a state alert at time unit t if and only if $\sum_{i=1}^{n} v_i(t-j) > T, \forall j \in [0, L-1]$.*

Proof 5 *In a filtering window of a node i, there may exist multiple periods of continuous local violation. We use $p_i^{'1}, p_i^{'2}, \ldots, p_i^{'k}$ to denote these periods of local violation where $k \in [1, p_i]$. Let $p_i^{'max} = \max\{p_i^{'1}, p_i^{'2}, \ldots, p_i^{'k}\}$ be the longest local violation period. Clearly, the filtering window τ_i($|\tau_i| = p_i$) contains $p_i^{'max}$, i.e. τ_i starts at least as early as $p_i^{'max}$ and ends at least as late as $p_i^{'max}$ does. We denote this inclusion relation as $p_i \succeq p_i^{'max}$.*

If constraints on T and L are violated, then there exists at least one series of local violation periods which overlap with each other and the total length of the overlapped period is L. For any one of such series $p_i^{'*}$, consider any one of its local violation periods p_i'. If p_i' is within one filtering window of node i, we have $p_i \succeq p_i'$. If p_i' spans multiple filtering windows, denoted as p_i^*, it is not hard to see $p_i^* \succeq p_i'$. Since it is the same for all p_i', all associated filtering windows, P_i^*, must satisfy $P_i^* \succeq p_i^{'*}$. As a result, a global poll is invoked no later than the state alert. The global poll sets κ to the length of observed $p_i^{'*}$. Similarly, subsequent global polls will keep increasing κ to the length of observed $p_i^{'*}$ until the last global poll which triggers the state alert at time t. The other direction can be proved in a similar way.

4.4.3 Communication Efficiency

Consider a state monitoring task with $n(n > 1)$ monitor nodes. Assume each T_i is perfectly tuned in the sense that one local violation occurs *if and only if* a global violation exists. Clearly, this is almost impossible in reality, as local violation does not always lead to global violation and global violation may correspond to multiple local violation. We use these "perfectly" tuned T_i to obtain the optimal performance of the instantaneous monitoring algorithm, so that we can study the lower bound of communication saving of the WISE algorithm. In addition, as Zipf distribution is often observed in distributed monitoring values[129], we assume the number of continuous local violation across nodes follows a Zipf distribution. Specifically, the probability of detecting continuous local violation of i time units is $Pr(x = i) = \frac{1}{H_{L+1}} \frac{1}{i+1}$, where H_{L+1} is the $(L + 1)^{th}$ Harmonic number defined by $H_{L+1} = \sum_{j=1}^{L+1} \frac{1}{j}$. Using Zipf distribution here is to simplify our analysis. In reality, continuous local violation needs not to follow this distribution. Furthermore, let the communication cost of local violation be 1 and that of global polls be n.

Theorem 7 *Given the above settings, let C_I be the communication cost of running the instantaneous monitoring algorithm with perfectly tuned T_i, and let C_W be the communication cost of running the WISE algorithm, which uses the same T_i and simply sets $p_i = 1$. The resulting gain in communication cost, given by $gain = \frac{C_I}{C_W}$, is $n(1 - \frac{1}{\log(L+1)})/(1 + \frac{n-2}{\log(L+1)})$*

Proof 6 *Since each local violation causes one global poll in the instantaneous triggering algorithm, we have $C_I = n \cdot \sum_{i=1}^{L} Pr(x = i) = n - \frac{n}{\log(L+1)}$. The communication cost of WISE consists of two parts, one is local violation, the other is global poll. Thus, $C_W = \sum_{i=1}^{L} Pr(x = i) + L \cdot (n-1) \cdot Pr(x = L)$. Therefore, the gain of using WISE is*

$$gain = \frac{C_I}{C_W} \quad \geqslant \quad \frac{n - \frac{n}{\log(L+1)}}{1 + \frac{n-2}{\log(L+1)}} \in [1, n)$$

The above theorem suggests that WISE yields more gain given larger L and n. For instance, when $L = 15$, $gain \geqslant \frac{3n}{n+3}$, the gain approximates to 3 when n is large enough. This implies that WISE scales well, which is confirmed by our experiment results. Furthermore, $gain$ is a theoretical bound derived with the unoptimized WISE algorithm. The actual gain is generally better (50% to 90% reduction in communication cost) with parameter tuning and optimized subroutines.

4.5 Scalable Parameter Tuning

The performance of WISE monitoring algorithm also depends on the setting of local monitoring parameters, i.e. T_i and p_i. To achieve the best communication efficiency, local monitoring parameters need to be tuned according to the given monitoring task and monitored value distributions. We first propose a centralized parameter tuning scheme which searches for the best parameters based on a sophisticated cost model. This scheme works well when the number of monitor nodes is moderate. However, datacenter environments often involve monitoring tasks running on a large number of nodes. The centralized scheme suffers from scalability issues in such large-scale monitoring tasks. First of all, the parameter space increases exponentially when the number of monitor nodes increases. As a result, the searching process of the centralized scheme may take considerable time to complete. Second, the centralized scheme requires the coordinator to collect monitored value distribution from all monitor nodes, which puts heavy burden on the coordinator node, especially with large-scale monitoring tasks.

To address these issues, we propose a scalable parameter tuning scheme which runs distributedly at each monitor node, and avoids searching in the entire parameter space and centralized data collection. In the rest of the section, we present detail of this distributed parameter tuning scheme.

4.5.1 Modeling Communication Cost

To begin with, we first introduce a cost model which can predict the communication cost of WISE monitoring algorithm given a set of monitoring parameters and the monitored value distribution. This model is frequently used for the development of our parameter tuning schemes.

4.5.1.1 Cost Analysis

Communication in the WISE algorithm consists of local violation reporting and global polls. We use C_l and $P_l(i)$ to denote the communication cost of sending a local violation report and the probability of sending it at one time unit on node i. Since a local violation report is of fixed size, we set C_l to 1. $P_l(i)$ is the probability of $v_i(t) > T_i$ and no local violation occurs during last p_i time units, because otherwise node i is in a filtering window during which it suppresses all violation reports.

Estimating the communication overhead for global polls is relatively complicated. To ease our discussion, we first define independent and continuous global polls:

Definition 7 *Given a global poll g occurring at time t, if there is no other global poll that occurs during time $[t - L + 1, t - 1]$, we say this global poll is an* independent global poll. *Otherwise, let g' be a global poll that happens during $[t - L + 1, t - 1]$, we say g' and g* overlap *with each other, denoted as $g \rightleftharpoons g'$. In addition, given a set of global polls, $\mathcal{G} = g_1, g_2, \ldots, g_k$, we refer $\mathcal{G}$ as a* continuous global poll *if $\forall g \in \mathcal{G}, \exists g' \in \mathcal{G}, g \rightleftharpoons g'$ and $\forall g'$ that $g' \rightleftharpoons g, g' \in \mathcal{G}$.*

Figure 27 shows an example of independent and continuous global polls. Intuitively, independent global polls are separated global polls which collect v_i values for L time units. Continuous global polls are adjoined global polls that each may collect v_i values for less than L time units, except the first one. In the following discussion, we refer a global poll which collects values for j time units as a j-windowed global poll. Clearly, $j = L$ for a independent global poll and $j > L$ for a continuous global poll. We use C_g^j to represent the cost associated with a j-windowed global poll. Since a j-windowed global poll requires all nodes to upload their v_i values of previous j time units, $C_g^j = n \cdot j$. In addition, we define P_g^j be the probability of a j-windowed global poll, since the probability of a global poll is also related with j.

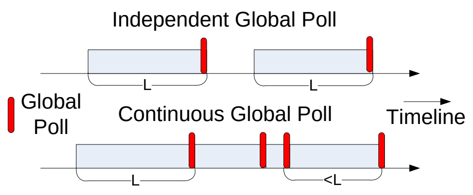


Figure 27: Independent and Continuous Global Polls

Given the above settings, the communication cost of the WISE algorithm can be estimated as follows:

$$C = \sum_{i=1}^{n} C_l P_l(i) + \sum_{j=L}^{\infty} C_g^j P_g^j \tag{5}$$

Note that C is essentially the expectation of communication cost for any time unit. Based on this cost function, we now define our parameter tuning problem as follows.

Problem Statement 4 *Given the global threshold T, monitoring window size L, and n monitor nodes, determine the values of $T_i, p_i, \forall i \in [1, n]$ so that the total communication cost C, given by Equation 5, is minimized.*

4.5.1.2 Determining Event Probabilities

We next present further detail on predicting the communication cost of WISE algorithm based on the cost function given by Equation 5. Clearly, we need to determine the probability of local violation events, $P_l(i)$, and the probability of global poll violation events, P_g^j, in order to compute C. Recall that $P_l(i)$ is the probability that a local violation occurs at time t and no local violation occurs during last p_i time units. Let V_i^t be the event of a violation on node i at time t, and correspondingly, $\overline{V_i^t}$ be the event of no violation on node i at time t. We have,

$$P_l(i) = P[\bigcap_{k=1}^{p_i} \overline{V_i^{t-k}}] \cdot P[V_i^t] \tag{6}$$

Compared with $C_l P_l(i)$, computing the cost for global polls is more complicated, as it depends on the states of monitor nodes. P_g^j is the probability that the size of a skeptical window equals to j. It is also the probability that at least one filtering window exists for each of the past j time units. Let W^t represent the event of at least one filtering window existing at t. Since W^t is independent among different t , we have, $P_g^j = P[\bigcap_{k=0}^{j-1} W^{t-k}] = (P[W^t])^j$. Denoting $P[W^t]$ by P_w, the cost of global polls is $\sum_{j=L}^{\infty} C_g^j P_g^j = n \sum_{j=L}^{\infty} j \cdot P_w^j$.

The sum part of the result is a variant of infinite geometric series, which can be solved via Taylor expansion. By solving this series, we have,

$$C_g(P_w) = \sum_{j=L}^{\infty} C_g^j P_g^j = n \frac{LP_w^L - (L-1)P_w^{L+1}}{(1-P_w)^2}$$

As the cost for global polls can be considered as a function of P_w, we use $C_g(P_w)$ to denote the cost of global polls. The value of P_w can be computed as,

$$P_w = 1 - P[\bigcap_{i=1}^{n} \bigcap_{k=1}^{p_i} \overline{V_i^{t-k}}] \tag{7}$$

This is because the probability of W^t is the probability of at least one node existing in its filtering window at time t. Up to this point, the only thing left unknown in both Equation 6 and 7 is the probability of $\overline{V_i^t}$, which depends on values of T_i, p_i and the distribution of v_i. To further compute $P_l(i)$ and P_w, we need to distinguish two types of stream values v_i. One is *time independent* values where v_i observed at the current moment is independent from those observed previously. The other type, *time dependent* values means v_i observed at the current moment is dependent from previous values. We next discuss the computation of $P_l(i)$ and P_w in both cases.

Time Independent v_i assumes v_i in different time units is i.i.d. In this case, $P[V_i^t]$ and $P[V_i^{t-1}]$ are independent. Thus, Equation 6 can be written as,

$$P_l(i) \quad = \quad (P[v_i \leqslant T_i])^{p_i}(1 - P[v_i \leqslant T_i]) \tag{8}$$

Similarly, Equation 7 now can be written as,

$$P_w \quad = \quad 1 - \prod_{i=1}^{n} (P[v_i \leqslant T_i])^{p_i} \tag{9}$$

Based on Equation 8 and 9, we only need the value of $P[v_i \leqslant T_i]$ to compute $P_l(i)$ and P_w. To obtain $P[v_i \leqslant T_i]$, each node maintains a histogram of the values that it sees over time as $H_i(x), x \in [0, T]$, where $H_i(x)$ is the probability of node i observing $v_i = x$. Given $H_i(x)$, $P[v_i \leqslant T_i] = \sum_{x=0}^{T_i} H_i(x)$.

Time Dependent v_i. We choose discrete-time Markov process, i.e. Markov chain, for modeling time dependent values, since it is simple and has been proved to be applicable to various real world stream data. Under this model, the values of future v_i and past v_i are independent, given the present v_i value. Formally, $P[v_i(t+1) = x | v_i(t) = x_t, \dots, v_i(1) = x_1] = P[v_i(t+1) = x | v_i(t) = x_t]$. For simplicity, we use v_i and v_i' to denote the present value and the value of the previous time unit respectively. Assuming v_i is time dependent, Equation 6 and 7 can be written as,

$$P_l(i) = P[v_i \leqslant T_i](P[v_i \leqslant T_i | v_i' \leqslant T_i])^{p_i - 1} P[v_i > T_i | v_i' \leqslant T_i] \tag{10}$$

$$P_w = 1 - \prod_{i=1}^{n} P[v_i \leqslant T_i](P[v_i \leqslant T_i | v_i' \leqslant T_i])^{p_i - 1} \tag{11}$$

To compute $P_l(i)$ and P_w, each monitor node maintains a set of transition probabilities $P[v_i = x | v_i' = x']$ where $x \in [0, T]$. Given these transition probabilities, $P[v_i \leqslant T_i] = \sum_{y=0}^{T} \sum_{x=0}^{T_i} P[v_i = x | v_i' = y]$, $P[v_i \leqslant T_i | v_i' \leqslant T_i] = \sum_{y=0}^{T_i} \sum_{x=0}^{T_i} P[v_i = x | v_i' = y]$ and $P[v_i > T_i | v_i' \leqslant T_i] = 1 - P[v_i \leqslant T_i | v_i' \leqslant T_i]$.

Interestingly, looking for the best values for T_i and p_i is essentially finding the best tradeoff between local violation and global polls which leads to the minimal communication cost. When increasing(decreasing) T_i, we reduce(increase) $P_l(i)$ which causes local violation to reduce(increase). However, larger(smaller) T_i also leads to larger(smaller) P_w which in turn increases(decreases) $C_g(P_w)$. It is also the same case for increasing or decreasing p_i.

4.5.2 Centralized Parameter Tuning

The centralized parameter tuning scheme is an intuitive development based on the above cost model. To determine best values for T_i and p_i, the centralized scheme adopts an EM-style local search scheme which iteratively looks for values leading to less cost. This scheme starts with two sets of initial values for T_i and p_i. Iteratively, it fixes one set of parameters and performs hill climbing to optimize the other set of parameters until reaching local minimum. It then fixes the optimized set and tunes the other one. It repeats this process until no better solution is found. To avoid local minimum, we run the scheme multiple times with different initial T_i and p_i values, and choose the best results.

4.5.3 Drawbacks of Centralized Tuning

The centralized scheme can find good local monitoring parameter values which minimizes communication given small number of monitor nodes. However, we find that this scheme suffers from scalability issues when this condition is not met.

First, as the centralized scheme holistically setting parameter for all nodes, the search space grows exponentially as the number of monitor nodes increases. Consequently, the search time of the centralized scheme also grows tremendously. Furthermore, when the number of monitor nodes is large, the search process causes significant consumption of CPU cycles at the coordinator, which could interfere with other jobs running on the coordinator node. One trade-off technique we apply to lower computation complexity is to reduce the search space by increasing the step size while performing hill climbing. This technique enables the centralized scheme to work with relatively large-scale monitoring tasks at the cost of less efficient parameters.

Second, the coordinator running the centralized scheme needs to collect the information of monitored value distribution, i.e. histograms in the time independent case and transition probabilities in the time dependent case, from all monitor nodes. This type of global information collecting is clearly not scalable and may consume considerable resources at the coordinator side. To address these issues, we propose a distributed parameter tuning scheme which allows each node to locally tune its monitoring parameters with minimal inter-node communication.

4.5.4 Distributed Parameter Tuning

The distributed parameter tuning scheme relieves the coordinator of the computation and communication burden by letting each node tune its monitoring parameters in a reactive manner based on events it observes. The main challenge in distributed parameter tuning is to effectively search for the best parameters at each monitor node without acquiring global information. We next describe detail of this scheme.

For ease of discussion, we use X_i to denote the probability of not having local violation at node i for both time dependent and independent v_i by defining X_i as following.

$$X_i = \begin{cases} P[v_i \leqslant T_i | v_i' \leqslant T_i] & \text{if time dependent} \\ P[v_i \leqslant T_i] & \text{if time independent} \end{cases}$$

By introducing X_i, Equation 5 can be written as follows

$$C = \sum_{i=1}^{n} C_l X_i^{p_i - 1}(1 - X_i) + C_g(1 - \prod_{i=1}^{n} X_i^{p_i})$$

Since X_i is the probability of no local violation, we assume $X_i \geqslant (1 - X_i)$ for reasonable monitoring applications. Thus,

$$C \leqslant \sum_{i=1}^{n} C_l X_i^{p_i} + C_g(1 - \prod_{i=1}^{n} X_i^{p_i})$$

Furthermore, Let $\alpha X_i \leqslant \min\{X_i | \forall i \in [1, n]\}$, where α can be predefined by user based on observed distribution, we have,

$$C \leqslant \sum_{i=1}^{n} C_l X_i^{p_i} + C_g(1 - (\alpha X_i)^{n p_i})$$

Let $Y_i = X_i^{p_i}$ and $\beta_i = \alpha^{n p_i}$, this relation can be written as,

$$C \leqslant \sum_{i=1}^{n} C_l Y_i + C_g(1 - \beta_i Y_i^n) \tag{12}$$

Thus, instead of directly tuning values for T_i and p_i, we can optimize values for Y_i. In fact, Y_i can be considered as the area of a 2-D "suppression window" at node i. The height of the window is controlled by X_i, which is determined by T_i, and the length of the window is controlled by p_i.

The distributed scheme adjusts Y_i at the monitor nodes based on their observed local violation reports and global poll events. Each local violation report from node i indicates that the area of the suppression window of node i is possibly lower than the optimum. Similarly, each global poll suggests the area of the suppression window is possibly higher than the optimum. Algorithm 1 shows the detail of the reactive scheme.

Algorithm 1 The Distributed Reactive Scheme

```
⋇ Invoked whenever received an event $E$
1: **if** $E$ = local violation **then**
2:     $Y_i \leftarrow \alpha Y_i$ with probability $\min(1, \frac{1}{\rho_i})$
3: **else** $\{E$ = global poll$\}$
4:     $Y_i \leftarrow \frac{Y_i}{\alpha}$ with probability $\min(1, \rho_i)$
5: **end if**
```

Choosing a proper value for ρ_i is critical for the reactive scheme to converge. Similar to the observation made in [57], the key point to achieve convergence is to make the scheme moves towards the optimal Y_i and stays at the optimal Y_i values once it reaches them. Assume the value of Y_i is not optimal, then either $Y_i < Y_i^{opt}$, which leads to $P_l(Y_i) > P_l(Y_i^{opt})$ and $P_w(Y) < P_w(Y^{opt})$, or $Y_i > Y_i^{opt}$, which leads to $P_l(Y_i) < P_l(Y_i^{opt})$ and $P_w(Y) > P_w(Y^{opt})$, where Y_i^{opt} is the optimal Y_i, Y and Y^{opt} stands for all Y_i and all Y_i^{opt} respectively. In the first case, we have $\frac{P_l(Y_i)}{P_w(Y)} > \frac{P_l(Y_i^{opt})}{P_w(Y^{opt})}$. By setting $\rho_i = \frac{P_w(Y^{opt})}{P_l(Y_i^{opt})}$, we have $\rho_i P_l(Y_i) > P_w(Y)$, which means the value of Y_i decreases. Similarly, we can see that the value of Y_i increases when $Y_i > Y_i^{opt}$. Thus, the reactive scheme reaches stable state when $\frac{P_l(Y_i)}{P_w(Y)} = \frac{P_l(Y_i^{opt})}{P_w(Y^{opt})}$.

While estimating the exact Y_i^{opt} is infeasible, we can still approximate this value by minimizing the upper bound of C based on Equation 12. More importantly, such computation can be done distributedly at each monitor node, as the right hand side of the equation can be divided into n items and each is only related with node i itself. Once each monitor node obtains its Y_i^{opt}, it sends this value to the coordinator. The coordinator gathers Y_i^{opt}

for all nodes and sends these values to all nodes. Each node then can compute its ρ_i based on the received Y_i values.

One remaining question is which component, T_i or p_i, to change when Y_i is updated. We develop the following heuristics to handle this problem. When Y_i is updated, node i first computes the new $T_i'(p_i')$ for the updated Y_i by using old $p_i(T_i)$. With probability $\min\{1, \Delta_{T_i}^{max}\frac{1}{T_i'-T_i}\}$, where $\Delta_{T_i}^{max}$ is the maximum step length for updating T_i, it updates p_i if $p_i' \leqslant L$. If p_i is not updated, it updates T_i if $T_i' \leqslant T$. The rationale is that T_i is restricted by the global threshold T, and thus, is updated only when the change is small.

To ensure correctness, when node i updates T_i, it sends T_i' and p_i' to the coordinator. If $T_i' < T_i$, the coordinator updates its slack $S \leftarrow T_i - T_i'$. Otherwise, the coordinator approves the update if $S \geqslant (T_i' - T_i)$. When $S < (T_i' - T_i)$, it notifies the node to update its T_i' to S if $S > 0$. If $S = 0$, it notifies the node to update p_i instead. Note that the above messages sent from monitor nodes can be combined with local violation reports or global poll messages, as an update is necessary only when a local violation or a global poll occurs.

4.6 *Performance Optimization*

The performance of WISE can be further optimized by improving the implementation of its major subroutines. In this section, we describe two interesting optimization techniques of this kind, one for enhancing the global poll procedure at the coordinator side and the other for improving local violation reporting procedure at the monitor node side.

4.6.1 Staged global polls

In the global poll procedure we introduced earlier, each node i sends its buffered $v_i(t-j)$ values, where $j \in [0, L]$, to the coordinator for state alert verifying. However, as the coordinator, more often than not, does not need all buffered values from all nodes to determine whether a state alert exists, such a global poll procedure usually causes unnecessary communication.

To further reduce the communication cost for global polls while still ensure the correctness of the monitoring algorithm, we propose a novel *staged global poll* procedure as an optimization technique. The staged global poll procedure divides the original global poll process into three stages. In each stage, only part of the $v_i(t-j), j \in [0, L]$ values are transmitted. In addition, if an early stage already rules out or triggers a state alert, then the rest of the stages can be avoided. Even if all stages are required, the new procedure transmits the same amount of v_i data as the original one.

Stage One. Node i only sends those $v_i(t-j)$ values that satisfies $v_i(t-j) \leqslant T_i$. Once received all the data, the coordinator tries to rule out the state alert by looking for a time unit t' in which $v_i(t') \leqslant T_i, \forall i \in [1, n]$. If such a time unit is found, it suggests that there exists at least one gap, i.e. a slot without violations, between local violations, and thus, the state alert can be ruled out.

Stage Two. If such gaps are not found, the global poll process enters the second stage, where it tries to confirm the existence of a state alert without invoking further communication. Specifically, the coordinator computes a partial slack $S'(t) = \sum_{i \in G(t)} T_i - v_i(t)$, where $G(t) = \{i | v_i(t) < T_i\}$ for all time units associated with the global poll. This partial slack $S'(t)$ is essentially the sum of "space", $T_i - v_i(t)$ at nodes not having local violation at time t. In addition, let $O = \{t | S'(t) \geqslant |N - G(t)|\}$ where N is the set of all monitor nodes and $N - G(t)$ is the set of nodes having local violations at time t. Note that $|N - G(t)|$ is the lower bound of the sum of "overflow", $v_i(t) - T_i$ at nodes having local violation at time t. The coordinator then triggers a state alert if $O = \emptyset$, because a state alert must exist if the sum of "space" is smaller than the sum of "overflow" for all time units associated with the global poll.

Final Stage. If the second stage does not trigger a state alert, the third, also the last, stage begins, in which the coordinator notifies all nodes that detected local violation at time units $t \in O$ (can be inferred based on data received in the first stage) to send the rest of their unsent v_i data. Once the data is received, the coordinator triggers a state alert if

$S'(t) < \Delta(t)$ for all t associated with the global poll, where $\Delta(t) = \sum_{i \in N-G(t)} v_i(t) - T_i$, it terminates the global poll procedure otherwise.

The correctness proof of the staged global poll is straightforward. The first stage *rules out* state alerts according to a sufficient condition of *not having* state alerts, i.e. the existence of "gaps". The second stage *triggers* state alerts according to a sufficient condition of *having* state alerts, i.e. the sum of "space" is smaller than the sum of "overflow" at all times units. Finally, the last stage collects all buffered values from all nodes, and thus, can always make correct decisions.

The staged global poll reduces significant amount of communication cost compared with the original global poll, as witnessed by our experiment results. The first and the second stages has the capability of ruling out or triggering state alerts with only a subset of buffered values in most cases. In the worse case, the staged global poll transmits the same amount of buffered values as the original global poll does, since the last stage only collects previously uncollected values.

4.6.2 Termination messages

In the original WISE algorithm, when a monitor node enters a filtering window, it no longer reports any local violation until it leaves the window. Although the coordinator assumes the node is experiencing local violations throughout its filtering window, it may not be true, as v_i may drop below T_i before the filtering window ends. While this scheme is very communication efficient, it also reduces the chance of ruling out global polls, as filtering windows may "exaggerate" the length of real local violation period and overlap with each other undesirably.

Based on this observation, we optimize the violation reporting procedure by letting node send a termination message which contains the information of un-violated time units at the end of a filtering window when it helps the coordinator to avoid global polls. While this modified scheme introduces extra communication at the end of filtering windows, the corresponding termination message, when properly generated, may avoid unnecessary global polls, and thus, reduces the total communication cost.

Specifically, a termination message of node i contains sequence numbers of time units during which $v_i(t) \leqslant T_i$. The sequence number here is associated with the previous violation report sent by node i, and thus, is defined over $(0, p_i - 1]$. When receiving a termination message m_t of a filtering window τ, the coordinator first updates τ by removing the time units contained in m_t, and then, use the updated filtering window to calculate its skeptical window.

Due to lack of global information, it is difficult for a monitor node to locally determine whether a termination message would help the coordinator to discover gaps between filtering windows. Clearly, always sending termination messages at the end of filtering window is not efficient. A termination message is beneficial only when the corresponding filtering window contains sufficient un-violated time units, because the more un-violated time units one termination message contains, the more likely the coordinator can avoid global polls. Therefore, we use $\frac{|m_t|}{p_i}$, where $|m_t|$ is the number of time units in the termination message m_t, to measure the likeliness of m_t can reveal gaps in the skeptical window. In our experiment, we restrict that a node i sends a termination message only when $\frac{|m_t|}{p_i} \geqslant 0.75$. By introducing this restriction, only nodes that observe adequate number of un-violated time units within its filtering windows send out termination messages.

4.7 Experimental Evaluation

We performed extensive experiments over both real world and synthetic traces to evaluate WISE. First, we evaluate the basic WISE with centralized parameter tuning. Our empirical study shows several important observations:

- WISE achieves a reduction from 50% to 90% in communication cost compared with instantaneous monitoring algorithm[57] and simple alternative schemes.
- The centralized parameter tuning scheme effectively improves the communication

efficiency.

- The optimization techniques further improve the communication efficiency of WISE.

Second, we evaluate the scalability of the WISE system with respect to different design choices and optimizations, especially we compare WISE equipped with the two optimizations with the basic WISE, and compare the improved WISE, powered by the distributed parameter tuning scheme, with the basic WISE using centralized tuning. We highlight the experimental results we observed as follows:

- WISE scales better than the instantaneous algorithm in terms of communication overhead. It scales even better with the distributed parameter tuning scheme.

- While the distributed parameter tuning scheme performs slightly worse than the centralized scheme, it scales better, and thus, is suitable for large scale distributed systems.

- The two optimization techniques continue to contribute additional communication saving when running with the distributed parameter tuning scheme.

4.7.1 Experiment Settings

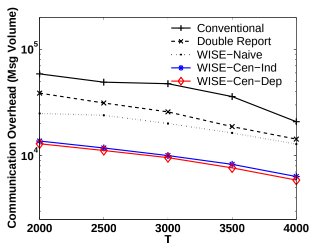
We consider our simulation scenario as detecting DDoS attacks for a set of distributed web servers. Each server is equipped with a monitoring probe. In addition, a centralized monitoring server watches the total number of HTTP requests received at different web servers. When the total number of requests continuously stays above a predefined threshold T for L time units, the monitoring server triggers a state alert.

We compare communication efficiency and scalability of the WISE algorithm, the instantaneous monitoring algorithm, and simple alternative window-based schemes. We choose the non-zero slack instantaneous monitoring algorithm[57] for comparison, as it is the most recent instantaneous monitoring approach and is reported to achieve significant communication reduction($\leqslant 60\%$) over previous approaches. The non-zero slack algorithm employs a local value threshold at each monitor node and reports local violation whenever the local value exceeds the local value threshold. It uses a set of optimization techniques to set optimal local threshold values so that $\sum T_i - T > 0$, a.k.a the slack may be positive.

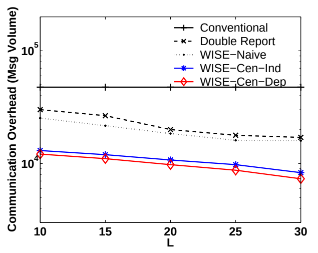
We also use several simple alternative window based state monitoring schemes as evaluation baseline. These schemes include the aforementioned "double reporting" scheme and WISE with naive parameter setting. Monitor nodes running the double reporting scheme report both the beginning and the end of a local violation period and the coordinator delays the global poll until necessary. The naive parameter setting simply sets $T_i = \frac{T}{n}$ and $p_i = \max\{\frac{L}{n}, 2\}$ for node i.

We measure communication cost by message volume (the total size of all messages) and message number. By default, we use message volume for comparison. In addition, we categorize messages into data messages and control messages. Data messages are those containing monitored values, e.g. local violation reports. The size of data message is m where m is the number of values encapsulated in the message. Control messages refer to all the other messages and their size is 1.

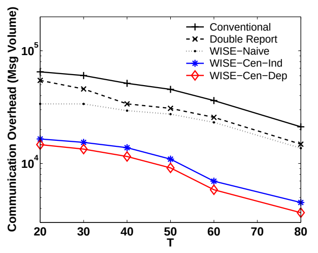
Our trace-driven simulator runs over two datasets. One dataset, *WorldCup*, contains real world traces of HTTP requests across a set of distributed web servers. The trace data comes from the organizers of the 1998 FIFA Soccer World Cup[18] who maintained a popular web site that was accessed over 1 billion times between April 30, 1998 and July 26, 1998. The web site was served to the public by 30 servers distributed among 4 geographic locations around the world. Thus, the traces of WorldCup provide us a real-world, large-scale distributed dataset. In our experiments, we used the server log data consisting of 57 million page requests distributed across 26 servers that were active during that period. We set the length of time unit to 1 minute and invoke a reconfiguration every 1,000 time units. Although results presented here are based on a 24-hour time slice (from 22:01:00 June 6th GMT to 22:00:00 June 7th GMT) of the system log data, we conducted a series of experiments over log data that spanned different days and different hours of day and we



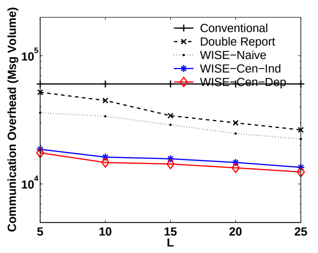
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



(c) Synthetic - Increasing T



(d) Synthetic - Increasing L

Figure 28: Comparison of Communication Efficiency in Terms of Message Volume

observed very similar results.

The other dataset, *Synthetic*, contains randomly generated traces that give us the freedom of evaluating parameters cannot be controlled in real world traces. For instance, we can increase the number of monitor nodes from 20 to 5000 for scalability evaluation. We first generate a trace of aggregate values and then distribute values to different nodes based on Uniform or Zipf distributions. Unless otherwise specified, the number of nodes is 20 and

Uniform distribution is applied. To track data distribution, we use equi-depth histograms at each monitor node and we also employ exponential aging on histograms to make it reflecting recent observed values more prominently than older ones. For both datasets, the parameter reconfiguration interval is 1000 time units.

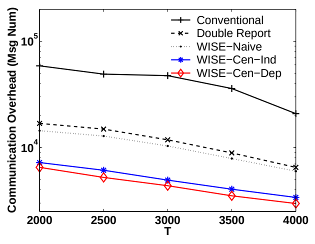
4.7.2 Results

4.7.2.1 Comparison of communication efficiency.

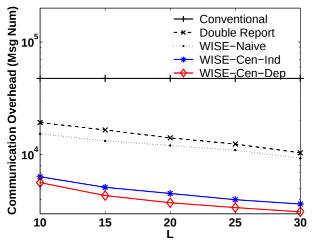
Figure 28 and 29 compare the communication overhead of WISE enhanced by centralized tuning(WISE-Cen) with that of the instantaneous monitoring algorithm(Instantaneous), the double reporting scheme(Double Report) and WISE with naive parameter setting(WISE-Naive) for the World Cup dataset and Synthetic dataset. We vary T and L in a way that the total length of global violation takes up from 0% to 50% of the total trace length. By default, we set $T = 2500(20)$ and $L = 15(10)$ for the WorldCup(Synthetic) dataset.

Figure 28(a) shows the total message volume generated by WISE is nearly a magnitude lower than that of the instantaneous approach. Double Report and WISE-Naive, while outperform the instantaneous approach as they delay global polls, generate more traffic compared with WISE. Double Report suffers from frequent reporting for short violation periods, especially when T is small. WISE-Naive fails to achieve better efficiency because it does not explore different value change patterns at different nodes. Note that parameter setting schemes using the time independent model(Ind) performs slightly better than those using time dependent one(Dep). However, as the time dependent model associates higher communication and computation cost, the time independent model is more desirable.

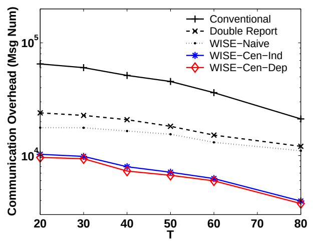
In Figure 28(b), while the instantaneous approach is not benefited from large values of L, the WISE algorithm pays less and less communication overhead as L grows, since nodes increase filtering window sizes and the coordinator rules out more global polls with increasing L. Figure 28(c) and 28(d) show similar results for the Synthetic dataset. Furthermore, as Figure 29 shows, WISE achieves even better efficiency advantage in terms



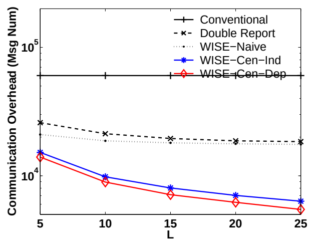
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



(c) Synthetic - Increasing T



(d) Synthetic - Increasing L

Figure 29: Comparison of Communication Efficiency in Terms of Message Number

of message *number*, as global polls in WISE collects multiple values, instead of a single value in the instantaneous approach. Thus, WISE is even more favorable when per message payload is insensitive to message sizes.

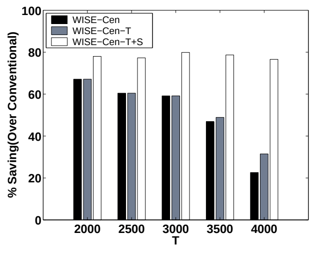
4.7.2.2 *Effect of optimization techniques.*

Figure 30(a) and 30(b) shows effect of *termination messages(T)* and *staged global polls(S)* in terms of communication reduction, where the Y axis is the percentage of message volume saved over the instantaneous scheme for the WorldCup dataset. In Figure 30(a), the WISE monitoring algorithm achieves 60% to 80% saving after optimization. However, the saving of unoptimized ones reduces as T grows because of two reasons. First, the instantaneous scheme also causes less communication as T grows. Second, with growing T, the portion of global poll communication increases(as suggested later by Figure 36(a)) due to reduced local violation, and the original global poll is very expensive. Termination messages achieve relatively less saving compared with staged global polls. In Figure 30(b), the saving increases when L grows, as larger L leads to larger p_i. Figure 30(c) and 30(d) show similar results for the Synthetic dataset.

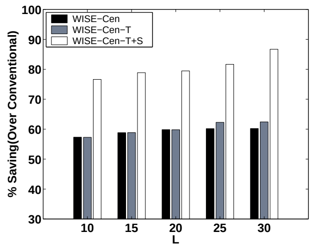
4.7.2.3 *Communication cost breakup analysis.*

Figure 31 shows communication cost breakup of WISE with centralized tuning, where communication overhead is divided into three parts: local violation reporting, global polls, and control. The first part is the overhead for value reporting in local violations, i.e. messages sent by nodes during local violations. The second part is the overhead for value reporting in global polls, which consists of messages with buffered stream values sent by nodes during global polls and notification messages from the coordinator. All the rest of the messages, most of which generated by the parameter tuning scheme, are classified as control messages. Furthermore, the left bar in each figure shows the percentage of different types of communication in message volume, and the right bar measures the percentage in message number.

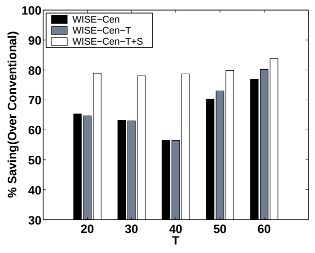
In Figure 31(a), as T grows, the portion of global poll communication steadily increases, as local violation occurs less frequently. The portion of control communication also increases, due to the reduction of local violation reporting and global polls. Similarly,



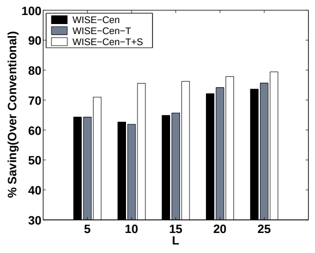
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



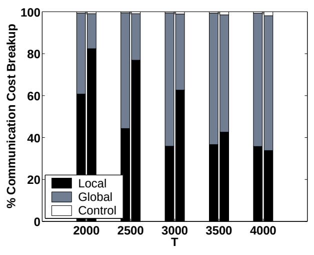
(c) Synthetic - Increasing T



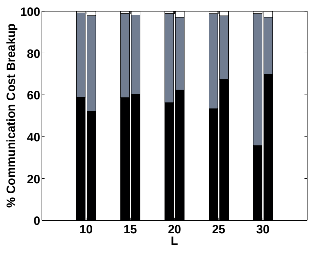
(d) Synthetic - Increasing L

Figure 30: Effectiveness of Optimization Techniques in Enhanced WISE(Message Volume)

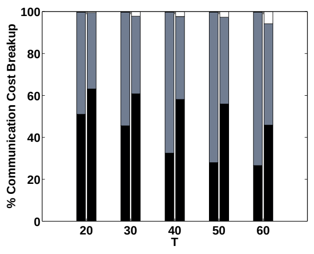
Figure 31(b) observes the growth of the global poll portion along with increasing L, as nodes increase p_i to filter more reports. Figure 31(c) and 31(d) provide similar results for the Synthetic dataset.



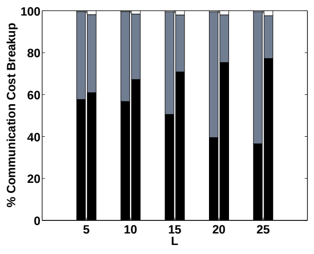
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



(c) Synthetic - Increasing T

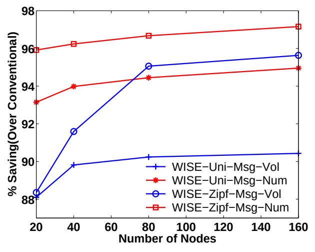


(d) Synthetic - Increasing L

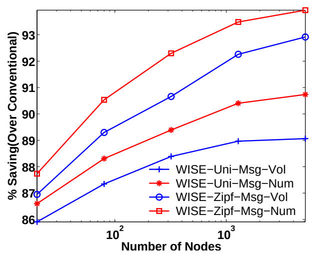
Figure 31: Communication Cost Breakup of WISE (Message Volume and Message Number)

4.7.2.4 *Scalability.*

Figure 32(a) and 32(b) evaluate the communication saving for WISE with centralized tuning. For World Cup dataset, we distributed the aggregated requests randomly to a set of 20 to 160 monitor nodes by Uniform and Zipf distributions. For the Synthetic dataset, we increased the number of nodes from 20 to 5000 nodes. When a Uniform distribution was used, every node received a similar amount of requests. When a Zipf distribution was assumed, a small portion of the nodes received most of the requests. For Zipf distribution, we chose a random Zipf exponent in the range of 1.0 to 2.0. Same as before, we measure communication cost in both message volume and message number.



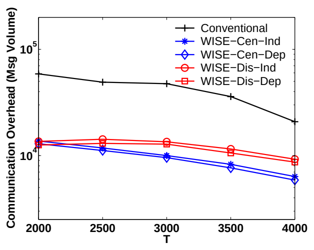
(a) WorldCup - Increasing n



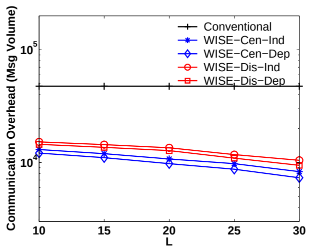
(b) Synthetic - Increasing n

Figure 32: Scalability (Message Volume and Number)

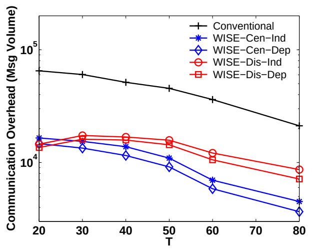
In Figure 32(a) and 32(b), the saving of WISE increases up to over 90% when the node number increases, which indicates that WISE scales better than the instantaneous approach. Interestingly, WISE performs better when Zipf distribution is used, because the parameter tuning scheme can set higher T_i and p_i to nodes observing higher values, which avoids considerable local violation and global polls. Again, WISE achieves higher advantage in communication reduction when we measure communication cost in number of messages



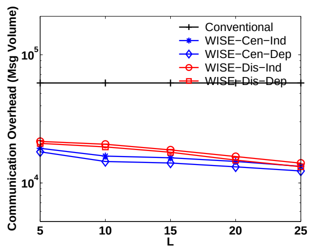
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



(c) Synthetic - Increasing T



(d) Synthetic - Increasing L

Figure 33: Comparison of Communication Efficiency in Terms of Message Volume

generated, as global polls in WISE collect values in multiple time units.

4.7.2.5 Distributed Tuning vs. Centralized Tuning

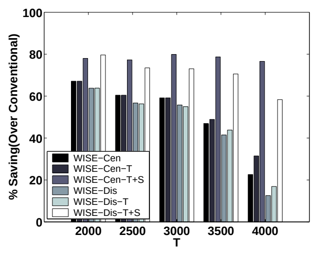
We now compare distributed tuning and centralized tuning in several different aspects. Figure 33 compares the communication efficiency achieved by the centralized tuning scheme and the distributed one. The centralized parameter tuning scheme(Cen) generally performs slightly better than the distributed one(Dis) does, as the centralized scheme has the complete value distribution information. Note that the distributed scheme works as good as the centralized scheme when T is relatively low, because violation is so frequent that there is little space for parameter optimization. When T increases, the centralized scheme starts to find better parameters than the distributed scheme does.

Another interesting observation is that the distributed scheme actually performs better than the centralized scheme when L is relatively large. As the centralized scheme overestimates the communication overhead for global polls, it tends to assign small values to p_i. The distributed scheme does not suffer from this problem as it can reactively increase p_i when it observes more local violations than global polls. As later proved in Figure 36(b), the centralized scheme pays much more local communication overhead than the distributed scheme does.

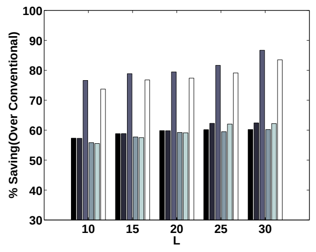
Figure 34 compares the effectiveness of the two optimization techniques when WISE is tuned by the centralized scheme and the distributed one respectively. In general, staged global poll and termination message work with both tuning schemes, although staged global poll achieves more communication reduction. Furthermore, as the distributed scheme tends to use larger p_i, WISE with distributed tuning encounters more global polls. Consequently, the staged global poll often gains more communication reduction when works with the distributed tuning scheme.

Figure 36 presents cost break(message volume) with centralized tuning scheme(left bar) and distributed tuning scheme(right bar). In Figure 36(a) and 36(b), the centralized scheme has a relatively larger portion of local violation overhead, as it overestimates the communication cost for global poll. In both figures, the distributed scheme pays more overhead in control, because it requires communication when adjusting local threshold T_i. Figure 36(c) and 36(d) provide similar results for the Synthetic dataset.

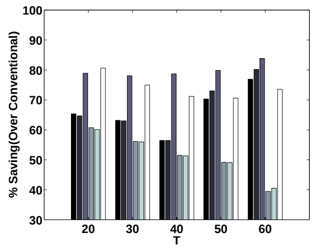
Figure 35(a) and 35(b) compares the scalability of centralized and distributed tuning schemes. The distributed scheme performs even better than the centralized scheme when



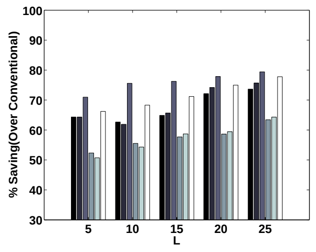
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



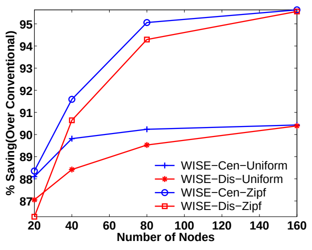
(c) Synthetic - Increasing T



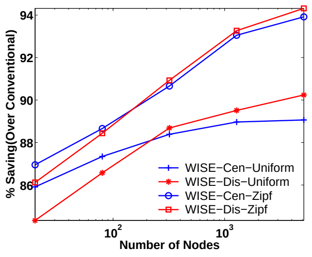
(d) Synthetic - Increasing L

Figure 34: Effectiveness of Optimization Techniques in Enhanced WISE(Message Volume)

the number of nodes is large. This is because we restrict the computation time used by the centralized scheme given a large number of nodes, which degrades its performance. Note that although computation and communication burden of the coordinator is not taken into account in the above results, it could cause serious workload imbalance between monitor nodes. For instance, in our experiment the CPU time consumed by the coordinator running



(a) WorldCup - Increasing n



(b) Synthetic - Increasing n

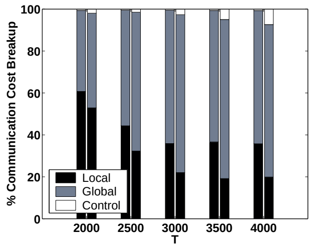
Figure 35: Scalability of Different Parameter Tuning Schemes

the centralized scheme is an order of magnitude more than that consumed by a monitor node under typical settings. Therefore, the distributed tuning scheme is a desirable alternative as it provides comparable communication efficiency and better scalability.

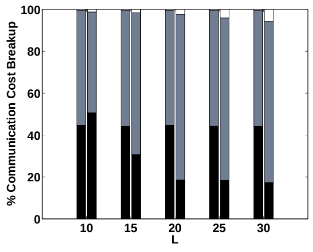
4.8 Related Work

Distributed data stream monitoring has been an active research area in recent years. Researchers have proposed algorithms for the continuous monitoring of top-k items[19], sums and counts [86] and quantiles[34], Problems addressed by these work are quite different from ours. While these work study supporting different operators, e.g. top-k and sums, over distributed data streams with guaranteed error bounds, we aims at detecting whether an aggregate of distributed monitored values violates constraints defined in value and time.

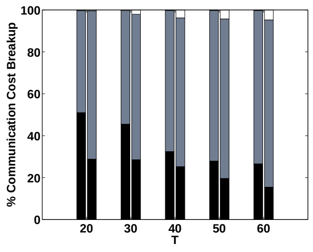
Recent work[39, 97, 60, 11, 57] on the problem of distributed constraint monitoring proposed several algorithms for communication efficient detection of constraint violation. They study a different problem by using an instantaneous monitoring model where a state alert is triggered whenever the sum of monitored values exceeds a threshold. By checking



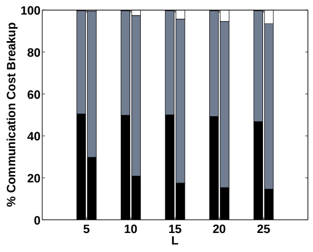
(a) WorldCup - Increasing T



(b) WorldCup - Increasing L



(c) Synthetic - Increasing T



(d) Synthetic - Increasing L

Figure 36: Communication Cost Breakup Between WISE with Centralized and Distributed Tuning Scheme

persistence, the window-based state monitoring model we study gains immunity to unpredictable bursty behavior and momentary outlier data [37], and provides more space for improving communication efficiency. In addition, the instantaneous model is a special case of ours when $L = 1$.

The early work[39] done by Dilman et al. propose a Simple Value scheme which sets all T_i to T/n and an Improved Value which sets T_i to a value lower than $T/n.$ Jain et al.[52] discusses the challenges in implementing distributed triggering mechanisms for network monitoring and they use local constraints of T/n to detect violation. The more recent work of Sharfman et al.[97] represents a geometric approach for monitoring threshold functions. Keralapura et al.[60] propose static and adaptive algorithms to monitor distributed sum constraints. Agrawal et al.[11] formulates the problem of selecting local constraints as an optimization problem which minimizes the probability of global polls. Kashyap et al. [57] proposes the most recent work in detecting distributed constraint violation. They use a non-zero slack scheme which is close to the idea of Improved Value scheme in [39]. They show how to set local thresholds to achieve good performance. We choose this non-zero slack scheme for comparison purpose.

The work that is perhaps closest to ours is that of Huang et al.[49]. They consider a variant of the instantaneous tracking problem where they track the cumulative amount of "overflows" which is $max\{0, \sum_i v_i - T\}$. This work makes two assumptions which may not be true: (1) All local threshold values are equal, and (2) local values follow a Normal distribution. In addition, it is unclear if the computed local thresholds in [49] optimize total communication costs. WISE employs a sophisticated cost model to estimate the total communication overhead and optimizes parameter setting based on this estimation. Furthermore, while [49] allows missed detections, WISE guarantees monitoring correctness.

Our approach aims at reducing unnecessary state alerts triggered by transient monitoring data outliers and noises by using filtering monitoring windows to capture continuous/stable violations. Wang and et. al. [119, 117] studied a series of entropy-based statistical techniques for reducing the false positive rate of detecting performance anomalies based on performance metric data such as CPU utilization with potential data noises. Compared with this work, our approach uses relatively simple noise filtering techniques rather than sophisticated statistical techniques to reduce monitoring false positives, but focuses on devising distributed window based state monitoring algorithms that minimize monitoring related communication. Note that our approach may still be used to efficiently collect monitoring data which are fed to statistical techniques for sophisticated application performance monitoring data analysis.

This chapter makes four contributions. First, we present the architectural design and deployment options for a WISE-enabled monitoring system. Second, we develop a novel distributed parameter tuning scheme that offers considerable performance improvement in terms of the scalability of the WISE framework. Third, we develop two concrete optimization techniques to further reduce the communication cost between a coordinator and its monitoring nodes. We show that both techniques guarantee the correctness of monitoring. Finally, we conduct a series of new experiments with a focus on how the distributed parameter tuning scheme and the two optimization techniques contribute to the enhancement of the scalability of WISE.

CHAPTER V

VOLLEY: VIOLATION LIKELIHOOD BASED STATE MONITORING FOR DATACENERS

5.1 Introduction

To ensure performance and reliability of distributed systems and applications, administrators often run a large number of monitoring tasks to continuously track the global state of a distributed system or application by collecting and aggregating information from distributed nodes. For instance, to provide a secured datacenter environment, administrators may run monitoring tasks that collect and analyze datacenter network traffic data to detect abnormal events such as Distributed Denial of Service (DDoS) attacks [42]. As another example, Cloud applications often rely on monitoring and dynamic provisioning to avoid violations to Service Level Agreements (SLA). SLA monitoring requires collecting of detailed request records from distributed servers that are hosting the application, and checks if requests are served based on the SLA (e.g., whether the response time for a particular type of requests is less than a given threshold). We refer to this type of monitoring as *distributed state monitoring* [94, 57, 79, 83] which continuously checks if a certain global state (e.g., traffic flowing to an IP) of a distributed system violates a predefined condition. Distributed state monitoring tasks are useful for detecting signs of anomalies and are widely used in scenarios such as resource provisioning [27], distributed rate limiting [94], QoS maintenance [81] and fighting network intrusion [65].

One substantial cost aspect in distributed state monitoring is the cost of collecting and processing monitoring data, a common procedure which we refer to as sampling. First, sampling operations in many state monitoring tasks are resource-intensive. For example, sampling in the aforementioned DDoS attack monitoring involves both packet logging and deep packet inspection over large amounts of datacenter network traffics [40]. Second, users of Cloud datacenter monitoring services (e.g. Amazon's CloudWatch) pay for monetary cost proportional to the frequency of sampling. Such monitoring costs can account for up to 18% of total operation cost [1]. Clearly, the cost of sampling is a major factor for the scalability and effectiveness of datacenter state monitoring.

Many existing datacenter monitoring systems provide periodical sampling as the only available option for state monitoring (e.g., CloudWatch [1]). Periodical sampling performs continuous sampling with a user-specified, fixed interval. It often introduces a cost-accuracy dilemma. On one hand, one wants to enlarge the sampling interval between two consecutive sampling operations to reduce sampling overhead. This, however, also increases the chance of mis-detecting state violations (e.g., mis-detecting a DDoS attack or SLA violation), because state violations may occur without being detected between two consecutive sampling operations with large intervals. On the other hand, while applying small sampling intervals lowers the chance of mis-detection, it can introduce significantly high resource consumption or monitoring service fees. In general, determining the ideal sampling interval with the best cost and accuracy trade-off for periodical sampling is difficult without studying task-specific characteristics such as monitored value distributions, violation definitions and accuracy requirements. It is even harder when such task-specific characteristics change over time, or when a task performs sampling over a distributed set of nodes.

We argue that one useful alternative to periodical sampling is a flexible monitoring framework where the sampling interval can be dynamically adjusted based on how likely a state violation will be detected. This flexible framework allows us to perform intensive monitoring with small sampling intervals when the chance of a state violation occurring is high, while still maintaining overall low monitoring overhead by using large intervals when the chance of a state violation occurring is low. As state violations (e.g., DDoS attacks and SLA violations) are relatively rare during the lifetime of many state monitoring tasks, this framework can potentially save considerable monitoring overhead, which is essential for Cloud state monitoring to achieve efficiency and high scalability.

In this chapter, we propose Volley, a violation likelihood based approach for distributed state monitoring. Volley addresses three key challenges in violation likelihood based state monitoring. First, we devise a sampling interval adaptation technique that maintains a user-specified monitoring accuracy while continuously adjusting the sampling interval to minimize monitoring overhead. Furthermore, this technique employs a set of low-cost estimation methods to ensure adaptation efficiency, and can flexibly support both basic and advanced state monitoring models. Second, for tasks that perform sampling over distributed nodes, we develop a distributed coordination scheme that not only ensures the global task-level monitoring accuracy, but also minimizes the total monitoring cost. Finally, we also propose a novel a-periodical state monitoring technique that leverages state-correlation to further reduce monitoring cost at the multi-task level.

We perform extensive experiments to evaluate the effectiveness of Volley with real world monitoring traces in a testbed datacenter environment running 800 virtual machines. Our results indicate that Volley reduces monitoring workload by up to 90% and still meets users' monitoring accuracy requirements at the same time.

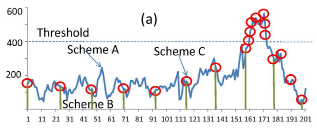
To the best of our knowledge, Volley is the first violation-likelihood based state monitoring approach that achieves both efficiency and controlled accuracy. Compared with other adaptive sampling techniques in sensor network [98, 29] and traffic monitoring [42, 96], our approach is also fundamentally different. First, sampling techniques in sensor networks often leverage the broadcast feature, while our system architecture is very different from sensor networks and does not have broadcast features. Second, while sensor networks usually run a single or a few tasks, we have to consider multi-task correlation in large-scale distributed environments. Third, some works (e.g. [29]) make assumptions on value distributions, while our approach makes no such assumptions. Finally, sampling in traffic monitoring is very different as these techniques perform packet/flow sampling to obtain a partial state of the traffic, while sampling on metric values is atomic. Furthermore, Volley is complementary to traffic sampling as it can be used together to offer additional cost savings by scheduling sampling operations.

5.2 Problem Definition

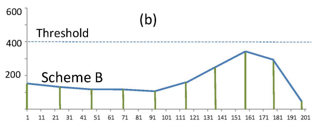
A distributed state monitoring task continuously tracks a certain global state of a distributed system and raises a state alert if the monitored global state violates a given condition. The global state is often an aggregate result of monitored values collected from distributed nodes. For example, assume there are n web servers and the i-th server observes a timeout request rate v_i (the number of timeout requests per unit time). A task may check if the total timeout request rate over all servers exceeds a given level T, i.e., check if $\sum v_i > T$ is satisfied. The general setting of a distributed state monitoring task includes a set of monitor nodes (e.g., the n web servers) and a coordinator. Each monitor can observe the current value of a variable (v_i) through sampling, and the coordinator aggregates local monitor values to determine if a violation condition is met (e.g., whether $\sum v_i > T$). We say a *state violation* occurs if the violation condition is met.

We define states based on the monitored state value and a user-defined threshold, which is commonly used in many different monitoring scenarios. The monitored state value can be a single metric value (e.g., CPU utilization). It can also be a scalar output of a function taking vector-like inputs, e.g., a DDoS detection function may parse network packets to estimate the likelihood of an attack. We assume that all distributed nodes have a synchronized wall clock time which can be achieved with the Network Time Protocol (NTP) at an accuracy of 200 microseconds (local area network) or 10 milliseconds (Internet) [92]. Hence, the global state can be determined based on synchronized monitored values collected from distributed nodes.

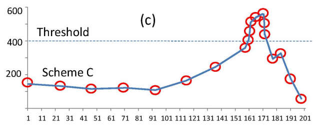
Existing works on distributed state monitoring focus on dividing a distributed state monitoring task into local tasks that can be efficiently executed on distributed nodes with



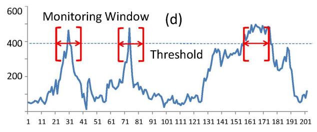
(a)



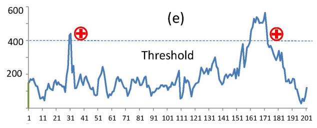
(b)



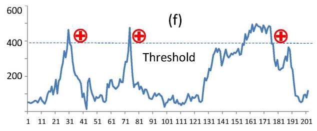
(c)



(d)



(e)



(f)

Figure 37: A Motivating Example

minimum inter-node communication [57, 83]. As a common assumption, a monitoring task employs a user specified sampling interval that is fixed across distributed nodes and over the entire task lifetime (more details in Section 6.5). In this chapter, we address a different yet important problem in distributed state monitoring, how to achieve efficient and accurate monitoring through dynamic sampling intervals.

5.2.1 A Motivating Example

DDoS attacks bring serious threats to applications and services running in datacenters. To detect DDoS attacks, some techniques [40] leverage the fact that most existing DDoS attacks lead to a growing difference between incoming and outgoing traffic volumes of the same protocol. For example, a SYN flooding attack often causes an increasing asymmetry of incoming TCP packets with SYN flags set and outgoing TCP packets with SYN and ACK flags set [40]. State monitoring based on such techniques watches the incoming rate of packets with SYN flags set P_i observed on a certain IP address, and the outgoing rate of packets with SYN and ACK flags set P_o observed on the same IP address. It then checks whether the traffic difference $\rho = P_i - P_o$ exceeds a given threshold, and if true, it reports a state alert. Such monitoring tasks collect network packets and perform deep packet inspection [38] in repeated monitoring cycles. For brevity, we refer to each cycle of packets collection and processing as sampling.

Periodical Sampling versus Dynamic Sampling. Sampling frequency plays a critical role in this type of DDoS attack monitoring. First, the sampling cost in the DDoS attack monitoring case is non-trivial. As we show later in the evaluation section, frequent collecting and analyzing packets flowing to or from virtual machines running on a server leads to an average of 20-34% server CPU utilization. As a result, reducing the sampling frequency is important for monitoring efficiency and scalability. Second, sampling frequency also determines the accuracy of monitoring. Figure 37 shows an example of a task which monitors the traffic difference between incoming and outgoing traffic for a given IP. The x-axis shows the time where each time point denotes 5 seconds. The y-axis shows the traffic difference ρ. The dash line indicates the threshold. We first employ high frequency sampling which we refer to as scheme A and show the corresponding trace with the curve in Chart (a). Clearly, scheme A records details of the value changes and can detect the state violation in the later portion of the trace where ρ exceeds the threshold. Nevertheless, the high sampling frequency also introduces high monitoring cost, and most of the earlier sampling yields little useful information (no violation). One may sample less frequently to reduce monitoring cost. For example, scheme B (bar) reduces considerable monitoring cost by using a relatively large monitoring interval. Consequently, as the curve in Chart (b) shows, scheme B also misses many details in the monitored values, and worst of all, it fails to detect the state violation (between the gap of two consecutive samples of scheme B). In general, it is difficult to find a fixed sampling frequency for a monitoring task that achieves both accuracy and efficiency.

One possible way to avoid this issue is to use dynamic sampling schemes where the sampling frequency is continuously adjusted on the fly based on the importance of the results. Ideally, such a scheme can use a low sampling frequency when the chance of detecting a violation is small, but when the chance is high, it can sample more frequently to closely track the state. Scheme C (circle) in Chart (c) gives an example of dynamic sampling. It uses a low sampling frequency at first, but switches to high frequency sampling when a violation is likely to happen.

While such an approach seems promising, realizing it still requires us to overcome several fundamental obstacles. First, we must find a way to *measure* and *estimate* violation likelihood before using it to adjust sampling intervals. The estimation should also be efficient. Second, since sampling introduces a trade-off between cost and accuracy, a dynamic sampling scheme should provide accuracy control by meeting a user-specified accuracy goal, e.g., "I can tolerate at most 1% state alerts being missed".

Advanced Monitoring Models. The previous example is based on an instantaneous monitoring model, where an alert is reported *whenever* a violation is detected. This model, however, often introduces unnecessary alerts with the presence of transient outliers and noises in monitored values. To address this issue, many real world monitoring tasks [1] use a window-based model where an alert is reported only when a period of *continuous violations* are detected. For instance, an SLA of a web service may define violations based on time windows, e.g., "the rate of timeout requests should not exceed 100 per second for more than 5 minutes". Chart (d) shows a window-based traffic monitoring example with 3 periods of violations. Although the first two violation periods last little time and are often considered as not important [83], the instantaneous model would still report them as state alerts. The window-based model, on the other hand, reports a state alert only for the last violation period, as the first two are shorter than the monitoring window (brackets).

Providing dynamic sampling for window-based state monitoring requires further study. Intuitively, under the instantaneous model, a state alert may happen when the monitored value is close to the threshold, but under the window-based model, state alerts are possible only when monitored values are continuously close to the threshold. Due to the differences in reporting conditions, the window-based model requires new violation likelihood estimation methods.

Distributed State Monitoring. When the monitored object is a set of servers hosting the same application, DDoS attack monitoring requires collecting traffic data of distributed servers. For example, suppose the traces in Chart (a) and (d) show the traffic difference on two distributed servers. The overall traffic difference on the two servers is the sum of trace values (denoted as v_1 and v_2) in Chart (a) and (d), and the monitoring task now checks if the overall traffic difference exceeds a global threshold T. For the sake of this example, we assume $T = 800$. While one can collect all packets on both servers (monitors) and send them to a coordinator which parses the packets to see if $v_1 + v_2 > T$, a more efficient way is to divide the task into local ones running on each monitor to avoid frequent communication. For example, we can assign monitors in Chart (a) and Chart (d) with local threshold $T_1 = 400$ and $T_2 = 400$ respectively. As a result, as long as $v_1 < T_1$ and $v_2 < T_2$, $v_1 + v_2 < T_1 + T_2 = T$ and no violation is possible. Hence, each server can perform local monitoring and no communication is necessary. We say a local violation occurs when a local threshold is exceeded, e.g. $v_1 > T_1$. Clearly, the coordinator only needs to collect values from both monitors to see if $v_1 + v_2 > T$ (referred to as a global poll) when a local violation happens.

The above local task based approach is employed in most existing state monitoring works [60, 83]. By dividing a global task into local ones, it also introduces new challenges for dynamic sampling. First, dynamic sampling schemes tune sampling intervals on individual monitors for a monitor-level accuracy requirement. For a task involve distributed monitors, how should we coordinate interval adjusting on each monitor to meet a task-level accuracy requirement? Second, suppose such coordination is possible. What principle should we follow to minimize the total monitoring cost? For instance, the trace in Chart (d) causes more local violations than Chart (a). Should we enforce the same level of accuracy requirement on both monitors?

State Correlation. Datacenter management relies on a large set of distributed state monitoring tasks. The states of different tasks are often related. For example, suppose the trace in Chart (e) shows the request response time on a server and the trace in Chart (f) shows the traffic difference on the same server. If we observe growing traffic difference in (f), we are also very likely to observe increasing response time in (e) due to workloads introduced by possible DDoS attacks. Based on state correlation, we can selectively perform sampling on some tasks only when their correlated ones suggest high violation likelihood. For instance, since increasing response time is a necessary condition of a successful DDoS attack, we can trigger high frequency sampling for DDoS attack monitoring only when the response time is high to reduce monitoring cost. Designing such a state-correlation based approach also introduces challenges. How to detect state correlation automatically? How to efficiently generate a correlation based monitoring plan to maximize cost reduction and minimize accuracy loss? These are all important problems deserving careful study.

5.2.2 Overview of Our Approach

Our approach consists of three dynamic sampling techniques at different levels. **Monitor Level Sampling** scheme dynamically adjusts the sampling interval based on its estimation of violation likelihood. The algorithm achieves controlled accuracy by choosing a sampling interval that makes the probability of mis-detecting violations lower than a user specified error allowance. Furthermore, we devise violation likelihood estimation methods with negligible overhead for both the instantaneous and the window-based state monitoring models. **Task Level Coordination** scheme is a lightweight distributed scheme that adjusts error allowance allocated to individual nodes in a way that both satisfies the global error allowance specified by the user, and minimizes the total monitoring cost. **Multi-Task Level State Correlation** based scheme leverages the state correlation between different tasks to avoid sampling operations that are least likely to detect violations across all tasks. It automatically detects state correlation between tasks and schedules sampling for different tasks at the datacenter level considering both cost factors and degree of state correlation. Due to space limitation, we refer readers to our technical report[76] for details of the state-correlation based monitoring techniques.

5.3 Accuracy Controlled Dynamic Sampling

A dynamic sampling scheme has several requirements. First, it needs a method to estimate violation likelihood in a timely manner. Second, a connection between the sampling interval and the mis-detection rate should be established, so that the dynamic scheme can strive to maintain a certain level of error allowance specified by users. Third, the estimation method should be efficient because it is invoked frequently to quickly adapt to changes in monitoring data. We next address these requirements and present the details of our approach.

5.3.1 Violation Likelihood Estimation

The specification of a monitoring task includes a default sampling interval I_d, which is the smallest sampling interval necessary for the task. Since I_d is the smallest sampling interval necessary, the mis-detection rate of violations is negligible when I_d is used. In addition, we also use I_d as our guideline for evaluating accuracy. Specifically, the specification of a monitoring task also includes an error allowance which is an acceptable probability of mis-detecting violations (compared with periodical sampling using I_d as the sampling interval). We use err to denote this error allowance. Note that $err \in [0, 1]$. For example, $err = 0.01$ means at most 1 percent of violations (that would be detected when using periodical sampling with the default sampling interval I_d) can be missed.

Recall that the instantaneous state monitoring model reports state alerts whenever $v > T$ where v is the monitored value and T is a given threshold. Hence, violation likelihood is defined naturally as follows,

Definition 8 *Violation Likelihood (under the instantaneous monitoring model) at time t is defined by $P[v(t) > T]$ where $v(t)$ is the monitored metric value at time t.*

Before deriving an estimation method for violation likelihood, we need to know 1) *when* to estimate and 2) for *which* time period the estimation should be made. For 1), because we want the dynamic sampling scheme to react to changes in monitored values as quickly as possible, estimation should be performed right after a new monitored value becomes available. For 2), note that uncertainties are introduced by unseen monitoring values between the current sampling and the next sampling (inclusive). Hence, estimation should be made for the likelihood of detecting violations within this period.

Violation likelihood for the next (future) sampled value is determined by two factors: the current sampled value and changes between two sampled values. When the current sampled value is low, a violation is less likely to occur before the next sampling time, and vice versa. Similarly, when the change between two continuously sampled values is large, a violation is more likely to occur before the next sampling time. Let $v(t_1)$ denote the current sampled value, and $v(t_2)$ denote the next sampled value (under current sampling frequencies). Let δ be the difference between the two continuously sampled values when the default sampling interval is used. We consider δ as a time-independent[1] random variable. Hence, the violation likelihood for a value $v(t_2)$ that is sampled i default sampling

[1]We capture the time-dependent factor with online statistics update which is described in Section 5.3.2.

intervals after $v(t_1)$ is,

$$P[v(t_2) > T] = P[v(t_1) + i\delta > T]$$

To efficiently estimate this probability, we apply Chebyshev's Inequality [45] to obtain its upper bound. The one-sided Chebyshev's inequality has the form $P(X - \mu \geqslant k\sigma) \leqslant \frac{1}{1+k^2}$, where X is a random variable, μ and σ are the mean and the variance of X, and $k > 0$. The inequality provides an upper bound for the probability of a random variable "digressing" from its mean by a certain degree, regardless of the distribution of X. To apply Chebyshev's inequality, we have

$$P[v(t_1) + i\delta > T] = P[\delta > \frac{T - v(t_1)}{i}]$$

Let $k\sigma + \mu = \frac{T-v(t_1)}{i}$ where μ and σ are the mean and variance of δ; we obtain $k = \frac{T-v(t_1)-i\mu}{i\sigma}$. According to Chebyshev's inequality, we have

$$P[\delta > \frac{T - v(t_1)}{i}] \leqslant 1/(1 + (\frac{T - v(t_1) - i\mu}{i\sigma})^2) \tag{13}$$

When selecting a good sampling interval, we are interested in the corresponding probability of mis-detecting a violation during the gap between two continuous samples. Therefore, we define the mis-detection rate for a given sampling interval I as follows,

Definition 9 *Mis-detection rate $\beta(I)$ for a sampling interval I is defined as $P\{v(t_1+\Delta_t) > T, \Delta_t \in [1, I]\}$ where $I(\geqslant 1)$ is measured by the number of default sampling intervals.*

Furthermore, according to the definition of $\beta(I)$, we have

$$\beta(I) = 1 - P\{\bigcap_{i\in[1,I]} (v(t_1) + i\delta \leqslant T)\}$$

Because δ is time-independent, we have

$$\beta(I) = 1 - \prod_{i\in[1,I]} (1 - P(v(t_1) + i\delta > T)) \tag{14}$$

According to Inequality 13, we have

$$\beta(I) \leqslant 1 - \prod_{i\in[1,I]} \frac{(\frac{T-v(t_1)-i\mu}{i\sigma})^2}{1 + (\frac{T-v(t_1)-i\mu}{i\sigma})^2} \tag{15}$$

Inequality 15 provides the method to estimate the probability of mis-detecting a violation for a given sampling interval I. We next present the dynamic sampling algorithm.

5.3.2 Violation Likelihood Based Adaptation

Figure 38 illustrates an example of violation likelihood based dynamic sampling. The dynamic sampling algorithm adjusts the sampling interval each time when it completes a sampling operation. Once a sampled value is available, it computes the upper bound of the mis-detection rate $\beta(I)$ according to inequality 15. We denote this upper bound with $\overline{\beta(I)}$. As long as $\beta(I) \leqslant \overline{\beta(I)} \leqslant err$ where err is the user-specified error allowance, the mis-detection rate is acceptable. To reduce sampling cost, the algorithm checks if $\overline{\beta(I)} \leqslant (1-\gamma)err$ for p continuous times, where γ is a constant ratio referred as the slack ratio. If true, the algorithm increases the current sampling interval by 1 (1 default sampling interval), i.e. $I \leftarrow I + 1$. The slack ratio γ is used to avoid risky interval increasing. Without γ, the algorithm could increase the sampling interval even when $\overline{\beta(I)} = err$, which is almost certain to cause $\overline{\beta(I+1)} > err$. Through empirical observation, we find that setting $\gamma = 0.2, p = 20$ is a good practice, and we consider finding optimal settings for γ and p as our future work. The sampling algorithm starts with the default sampling interval I_d, which is also the smallest possible interval. In addition, users can specify the maximum sampling interval denoted as I_m, and the dynamic sampling algorithm would never use a sampling interval $I > I_m$. If it detects $\overline{\beta(I)} > err$, it switches the sampling interval to the default one immediately. This is to minimize the chance of mis-detecting violations when the distribution of δ changes abruptly.

Because we use the upper bound of violation likelihood to adjust sampling intervals and the Chebyshev bound is quite loose, the dynamic sampling scheme is conservative on employing large sampling intervals unless the task has very stable δ distribution or the monitored values are consistently far away from the threshold. As sampling cost reduces sub-linearly with increasing intervals ($1 \rightarrow \frac{1}{2} \rightarrow \frac{1}{3} \cdots$), being conservative on using large

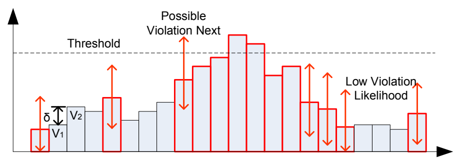


Figure 38: Violation Likelihood Based Adaptation

intervals does not noticeably hurt the cost reduction performance, but reduces the chance of mis-detecting important changes between sampling.

Since computing inequality 15 relies on the mean and the variance of δ, the algorithm also maintains these two statistics based on observed sampled values. To update these statistics efficiently, we employ an online updating scheme[61]. Specifically, let n be the number of samples used for computing the statistics of δ, μ_{n-1} denote the current mean of δ and μ_n denote the updated mean of δ. When the sampling operation returns a new sampled value $v(t)$, we first obtain $\delta = v(t) - v(t-1)$. We then update the mean by $\mu_n = \mu_{n-1} + \frac{\delta - \mu_{n-1}}{n}$. Similarly, let σ_{n-1} be the current variance of δ and σ_n be the updated variance of δ; we update the variance by $\sigma_n^2 = \frac{(n-1)\sigma_{n-1}^2 + (\delta - \mu_n)(\delta - \mu_{n-1})}{n}$. Both updating equations are derived from the definition of mean and variance respectively. The use of online statistics updating allows us to efficiently update μ and σ without repeatedly scanning previous sampled values. Note that sampling is often performed with sampling intervals larger than the default one. In this case, we estimate $\hat{\delta}$ with $\hat{\delta} = (v(t) - v(t-I))/I$, where I is the current sampling interval and $v(t)$ is the sampled value at time t, and we use $\hat{\delta}$ to update the statistics. Furthermore, to ensure the statistics represent the most recent δ distribution, the algorithm periodically restarts the statistics updating by setting $n = 0$ when $n > 1000$.

5.3.3 Window Based State Monitoring

Window based state monitoring is widely used in real world monitoring tasks [83] and can even be the only monitoring model for some cloud monitoring services[1] due to its ability to filter transient outliers. Recall that window based monitoring reports an alert only when it detects a period of continuous violations lasting at least w default sampling intervals I_d where $w > 1$. While one may apply the techniques we developed above to enable dynamic sampling in window based state monitoring, such an approach is not the most efficient one. As the stricter alert condition in the window based model often results in lower violation likelihood compared with that in the instantaneous model, an efficient scheme should leverage this to further reduce cost.

When dynamic sampling causes the mis-detecting of an alert under the window based monitoring model, a set of conditions have to be met. First, there must be a period of continuous violations with a length of at least wI_d after the current sampling procedure. Second, the following sampling procedure detects at most $w - 1$ continuous violations. Otherwise, the algorithm would have detected the alert. Accordingly, the probability of missing an alert P_{miss} after a sampling operation at time t_0 is,

$$P_{miss} = P\{|V| \geqslant w \bigcap t_0 \leqslant t_s(V) < t_e(V) \leqslant t_0 + I + w - 1\}$$

where V denotes a period of continuous violations, $|V|$ is the length of the continuous violations (measured by I_d), and $t_s(V)$ and $t_e(V)$ are the start time and end time of V. To get a closed form, we could reuse Equation 14 and consider all possible situations that could trigger a state alert event. However, doing so would cause expensive computation, as it involves $I - 1$ cases of continuous violations with length w, each of which requires further estimation of its probability. To avoid high estimation cost, we approximate P_{miss} with its upper bound. Let v_i be the monitored value at time i; we have,

$$P_{miss} \leqslant P\{\sum_{i \in [t_0, t]} v_i \geqslant wT, t = t_0 + I + w - 1\}$$

where T is the threshold of the monitoring task. The rationale is that for continuous violations of length w to occur, at least w sampling values should be larger than T. Since $v(i + 1) = v(i) + \delta$, we have

$$P_{miss} \leqslant P\{(I + w - 1)(v_0 + \delta(I + w)/2) \geqslant wT\}$$

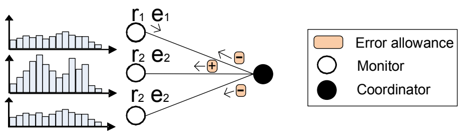


Figure 39: Distributed Sampling Coordination

, which leads to

$$P_{miss} \leqslant P\{\delta \geqslant \frac{2(wT - v_0(I + w - 1))}{(I + w)(I + w - 1)}\} \tag{16}$$

The righthand side of the inequality can be bounded with Chebyshev's inequality based on the mean and variance of δ similar to the estimation for the instantaneous model. Hence, we can control the monitoring accuracy by choosing a monitoring interval I satisfying $P_{miss} \leqslant err$ where err is the error allowance. We can simply substitute inequality 15 with inequality 16 in the monitoring algorithm to support efficient window based monitoring.

Note that sampling in window based monitoring requires to collect values within the current sampling interval I to measure the length of continuous violations. This is different from sampling in instantaneous monitoring where only the value at the sampling time is needed. As we show in the evaluation section, window-based violation estimation offers more cost saving compared with instantaneous violation estimation does in window-based monitoring when per-sampling (versus per-value) cost is dominant (e.g., cloud monitoring services that charge based on sampling frequency [1] and high per-message overhead [79])

5.4 Distributed Sampling Coordination

A distributed state monitoring task performs sampling operations on multiple monitors to monitor the global state. For dynamic sampling, it is important to maintain the user-specified task-level accuracy while adjusting sampling intervals on distributed monitors.

5.4.1 Task-Level Monitoring Accuracy

Recall that a distributed state monitoring task involves multiple monitors and a coordinator. Each monitor performs local sampling and checks if a local condition is violated. If true, it reports the local violation to the coordinator which then collects all monitored values from all monitors to check if the global condition is violated.

The local violation reporting scheme between monitors and the coordinator determines the relation between local monitoring accuracy and global monitoring accuracy. When a monitor mis-detects a local violation, the coordinator may miss a global violation if the mis-detected local violation is indeed part of a global violation. Let β_i denote the mis-detection rate of monitor i and β_c denote the mis-detection rate of the coordinator. Clearly, $\beta_c \leqslant \sum_{i \in N} \beta_i$ where N is the set of all monitors. Therefore, as long as we limit the sum of monitor mis-detection rates to stay below the specified error allowance err, we can achieve $\beta_c \leqslant \sum_{i \in N} \beta_i \leqslant err$.

5.4.2 Optimizing Monitoring Cost

For a given error allowance err, there are different ways to distribute err among monitors, each of which may lead to different monitoring cost. For example, suppose trace (e) and (f) in Figure 37 show values monitored by monitor 1 and 2. As (f) is more likely to cause violations than (e), when evenly dividing err among monitor 1 and 2, one possible result is that $I_1 = 4$ (interval on monitor 1) and $I_2 = 1$ (interval on monitor 2). The total cost reduction is $(1 - 1/4) + (1 - 1) = 3/4$. When assigning more err to monitor 2 to absorb frequent violations, we may get $I_1 = 3, I_2 = 2$ and the corresponding cost reduction is $2/3 + 1/2 = 7/6 > 3/4$. Therefore, finding the optimal way to assign err is critical to reduce monitoring cost.

Nevertheless, finding the optimal assignment is difficult. Brute force search is impractical ($O(n^m)$ where m is the number of monitors and n is the number of minimum assignable units in err). Furthermore, the optimal assignment may also change over time when characteristics of monitored values on monitors vary. Therefore, we develop an iterative scheme that gradually tunes the assignment across monitors by moving error allowance from monitors with low cost reduction yield (per assigned err) to those with high cost reduction yield.

Figure 39 illustrates the process of distributed sampling coordination. The coordinator first divides err evenly across all monitors of a task. Each monitor then adjusts its local sampling interval according to the adaptation scheme we introduced in Section 5.3 to minimize local sampling cost. Each monitor i locally maintains two statistics: 1) r_i, potential cost reduction if its interval increased by 1 which is calculated as $r_i = 1 - \frac{1}{I_i+1}$; 2) e_i, error allowance needed to increase its interval by 1 which is calculated as $e_i = \frac{\overline{\beta(I_i)}}{1-\gamma}$ (derived from the adaptation rule in Section 5.3.2).

Periodically, the coordinator collects both r_i and e_i from each monitor i, and computes the cost reduction yield $y_i = \frac{r_i}{e_i}$. We refer to this period as an *updating period*, and both r_i and e_i are the average of values observed on monitors within an updating period. y_i essentially measures the cost reduction yield per unit of error allowance. After it obtains y_i from all monitors, the coordinator performs the following assignment $err'_i = err\frac{y_i}{\sum_i y_i}$, where err'_i is the assignment for monitor i in the next iteration. Intuitively, this updating scheme assigns more error allowance to monitors with higher cost reduction yield. The tuning scheme also applies throttling to avoid unnecessary updating. It avoids reallocating err to a monitor i if $err_i < \underline{err}$ where constant $\underline{err}$ is the minimum assignment. Furthermore, it does not perform reallocation if $\max\{y_i/y_j, \forall i, j\} < 0.1$. We set the updating period to be every thousand I_d and $\underline{err}$ to be $\frac{err}{100}$.

5.5 *Evaluation*

We deploy a prototype of Volley in a datacenter testbed consisting of 800 virtual machines (VMs) and evaluate Volley with real world network, system and application level monitoring scenarios. We highlight some of the key results below:

- The violation likelihood based adaptation technique saves up to 90% sampling cost for both instantaneous and window based models. The accuracy loss is smaller or close to the user specified error allowances.

- The distributed sampling coordination technique optimizes the error allowance allocation across monitors and outperforms alternative schemes.

5.5.1 Experiment Setup

We setup a virtualized datacenter testbed containing 800 VMs in Emulab [120] to evaluate our approach. Figure 40 illustrates the high-level setup of the environment. It consists of 20 physical servers, each equipped with a 2.4 GHz 64bit Quad Core Xeon E5530 processor, 12 GB RAM and runs XEN-3.0.3 hypervisor. Each server has a single privileged VM/domain called Domain 0 (`Dom0`) which is responsible for managing the other unprivileged VMs/user domains [47]. In addition, each server runs 40 VMs (besides `Dom0`) configured with 1 virtual CPU and 256MB memory. All VMs run 64bit CentOS 5.5. We implemented a virtual network to allow packet forwarding among all 800 VMs with XEN route scripts and `iptables`.

We implemented a prototype of Volley which consists of three main components: agents, monitors and coordinators. An agent runs within a VM and provides monitoring data. Agents play an important role in emulating real world monitoring environments. Depending on the type of monitoring, they either generate network traffic according to pre-collected network traces or provide pre-collected monitoring data to monitors when requested (described below). For each VM, a monitor is created in `Dom0`. Monitors collect monitoring data, process the data to check whether local violations exist and report local violations to coordinators. In addition, they also perform local violation likelihood based sampling adaptation described earlier. A coordinator is created for every 5 physical servers. They process local violation reports and trigger global polls if necessary. Furthermore, we also implement distributed sampling coordination on coordinators. We next present details on monitoring tasks.

Network level monitoring tasks emulate the scenario of detecting distributed denial of service (DDoS) attacks (Section 5.2.1) in a virtualized datacenter. We perform traffic

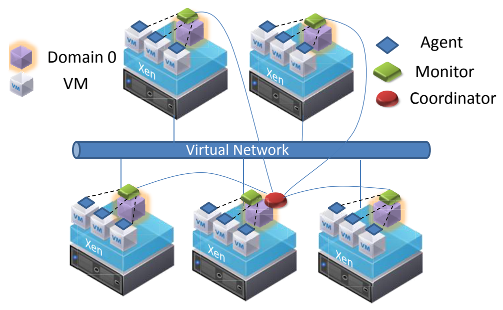
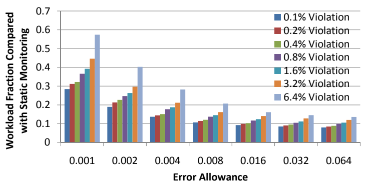


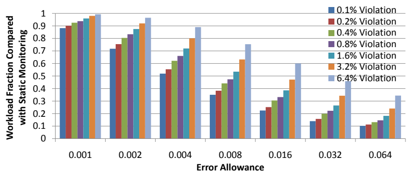
Figure 40: Experiment Setup

monitoring on each server (`Dom0`), rather than network switches and routers, because only `Dom0` can observe communications between VMs running on the same server. For a task involving a set V of VMs , their corresponding monitors perform sampling by collecting and processing traffic associated with the VM $v \in V$ (within a 15-second interval) to compute the traffic difference $\rho_v = P_i(v) - P_o(v)$ where $P_i(v)$ and $P_o(v)$ are the incoming number of packets with SYN flags set and the outgoing number of packets with both SYN and ACK flags set respectively. The sampling is implemented with `tcpdump` and bash scripts and the default sampling interval is 15 seconds. When ρ_v exceeds the local threshold, the monitor reports a local violation to its coordinator which then communicates with monitors associated with other VMs $V - v$ to check if $\sum_{v \in V} \rho_v > T$.

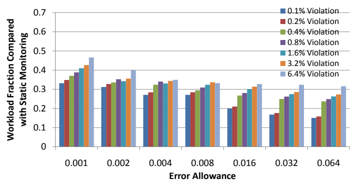
We port real world traffic observed on Internet2 network[2], a large-scale high-capacity network connecting research centers, into our testbed environment. The traces are in netflow v5 format and contain approximately 42,278,745 packet flows collected from 9 core routers in the Internet2 network. A flow in the trace records the source and destination IP addresses as well as the traffic information (total bytes, number of packets, protocol, etc.) for a flow of observed packets. We uniformly map addresses observed in netflow logs into VMs in our testbed. If a packet sent from address A to B is recorded in the logs, VM X (where A is mapped to) also sends a packet to VM Y (where B is mapped to). As netflow does not record SYN and ACK flags, we set SYN and ACK flags with a fixed probability $p = 0.1$ to each packet a VM sends. Note that ρ is not affected by the value of p as p has



(a) Network Level Monitoring



(b) System Level Monitoring



(c) Application Level Monitoring

Figure 41: Monitoring Overhead Saving under Different Error Allowance and State Alert Rates

the same effect to both P_i and P_o. In addition, let F denote the number of packets in a recorded flow. We also scale down the traffic volume to a realistic level by generating only F/n packets for this flow where n is the average number of addresses mapped to a VM.

System level monitoring tasks track the state of OS level performance metrics on VMs running in our testbed. A system level task triggers a state alert when the average value of a certain metric exceeds a given threshold, e.g. an alert is generated when the average memory utilization on VM-1 to VM-10 exceeds 80%. To emulate VMs in a production environment, we port a performance dataset[126] collected from hundreds of Planetlab[91] nodes to our VMs. This dataset contains performance values on 66 system metrics including available CPU, free memory, virtual memory statistics(vmstat), disk usage, network usage, etc. To perform sampling, a monitor queries its assigned VM for a certain performance metric value and the agent running inside the VM responds with the value recorded in the dataset. The default sampling interval is 5 seconds.

Application level monitoring tasks watch the throughput state of web applications deployed in datacenters. For example, Amazon EC2 can dynamically add new server instances to a web application when the monitored throughput exceeds a certain level[1]. We port traces of HTTP requests (> 1 billion) collected from a website hosted by a set of 30 distributed web servers[3]. Similar to system level monitoring, agents running on VMs respond with web server access logs in the dataset when queried by monitors so that they mimic VMs running a web application. The default sampling interval is 1 second.

Thresholds. Monitoring datasets used in our experiment are not labeled for identifying state violations. Hence, for a state monitoring task on metric m, we assign its monitoring threshold by taking $(100 - k)$-th percentile of m's values. For example, when $k = 1$, a network-level task reports DDoS alerts if $\rho > Q(\rho, 99)$ where $Q(\rho, 99)$ is the 99th percentile of ρ observed through the lifetime of the task. Similarly, when $k = 10$, a system-level task report state alerts if memory utilization $\mu > Q(\mu, 90)$. We believe this is a reasonable way to create state monitoring tasks as many state monitoring tasks try to detect a small percentage of violation events. In addition, we also vary the value of selectivity parameter k to evaluate the impact of selectivity in tasks.

5.5.2 Results

Monitoring Efficiency. Figure 41(a) illustrates the results for our network monitoring experiments where each task checks whether the traffic difference ρ on a single VM exceeds a threshold set by the aforementioned selectivity k (we illustrate results on distributed monitoring tasks (multiple VMs) later in Figure 44). We are interested in the ratio of sampling operations (y-axis) performed by Volley over those performed by periodical sampling (with interval I_d). We vary both the error allowance (x-axis) and the alert selectivity k in monitoring tasks (series) to test their impact on monitoring efficiency. Recall that the error allowance specifies the maximum percentage of state alerts allowed to be missed. An error allowance of 1% means that the user can tolerant at most 1% of state alerts not being detected. We see that dynamic sampling reduces monitoring overhead by 40%-90%. Clearly, the larger the error allowance, the more sampling operations Volley can save by reducing monitoring frequency. The alert state selectivity k also plays an important role, e.g. varying k from 6.4% to 0.1% can lead to 40% cost reduction. Recall that $k = 6.4$ means that 6.4% of monitored values would trigger state alerts. This is because lower k leads to fewer state alerts and higher thresholds in monitoring tasks, which allows Volley to use longer monitoring intervals when previous observed values are far away from the threshold (low violation likelihood). Since real world tasks often have small k, e.g. a task with a 15-second monitoring interval, generating one alert event per hour leads to a $k = 1/240 \approx 0.0042$. We expect Volley to save considerable overhead for many monitoring tasks.

Figure 41(b) shows the results for system level monitoring, where each monitoring task checks if the value of a single metric on a certain VM violates a threshold chosen by the aforementioned selectivity k. The results suggest that Volley also effectively reduces the monitoring overhead, with relatively smaller cost saving ratios compared with the network monitoring case. This is because changes in traffic are often less than changes in system metric values (e.g. CPU utilization). This is especially true for network traffic observed at night.

We show the results of application level monitoring in Figure 41(c) where each task checks whether the access rate of a certain object, e.g. a video or a web page, on a certain VM exceeds the k-determined threshold by analyzing the recent access logs on the VM. We observe similar cost savings in this figure. The high cost reduction achieved in the application level monitoring is due to the bursty nature of accesses. It allows our adaptation to use large monitoring intervals during off-peak times. We believe that our techniques can provide substantial benefits when this type of change pattern occurs in many other applications (e.g. e-business websites) where diurnal effects and bursty request arrival are common.

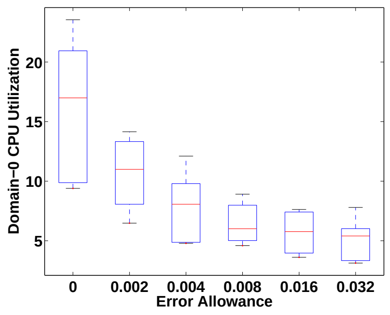


Figure 42: CPU Utilization

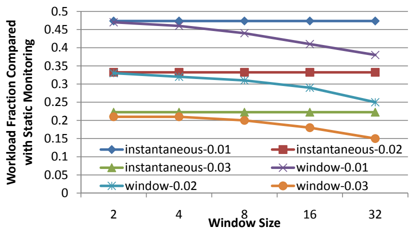


Figure 43: Window Based Model

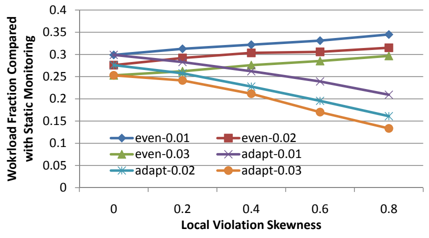


Figure 44: Distributed Coordination

Figure 42 uses box plots to illustrate the distribution of Dom0 CPU resource consumption (percentage) caused by network-level monitoring tasks with increasing error allowance. The upper and lower bound of boxes mark the 0.75 and 0.25 quantile. The line inside the box indicates the median. The whiskers are lines extending from each end of the box to show the extent of the rest of the utilization data. CPU resources are primarily consumed by packet collection and deep packet inspection, and the variation in utilization is caused by network traffic changes. When the error allowance is 0, our violation-likelihood based sampling is essentially periodical sampling and introduces fairly high CPU utilization (20-34%) which is prohibitively high for Dom0. This is because Dom0 needs to access hardware on behalf of all user VMs, and IO intensive user VMs may consume lots of Dom0

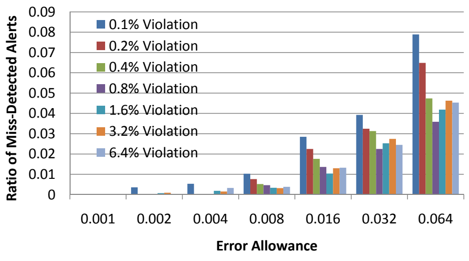


Figure 45: Actual Mis-Detection Rates

CPU cycles. When `Dom0` is saturated with monitoring and IO overhead, all VMs running on the same server experience seriously degraded IO performance[47]. With increasing error allowance, our approach quickly reduces the CPU utilization by at least a half and substantially improves the efficiency and scalability of monitoring.

Although system/application level monitoring tasks incur less overhead compared with the network monitoring case, Volley can still save significant monitoring cost when such tasks are performed by monitoring services that charge users based on sampling frequency[1]. Furthermore, the aggregated cost of these tasks is still considerable for datacenter monitoring. Reducing sampling cost not only relieves resource contention between application and infrastructure management, but also improves the datacenter management scalability[81].

Monitoring Accuracy. Figure 45 shows the actual mis-detection rate (y-axis) of alerts for the system-level monitoring experiments. We see that the actual mis-detection rate is lower than the specified error allowance in most cases. Among different state monitoring tasks, those with high alert selectivity often have relatively large mis-detection rates. There are two reasons. First, high selectivity leads to few alerts which reduces the denominator in the mis-detection rate. Second, high selectivity also makes Volley prefer low frequency which increases the chance of missing alerts. We do not show the results on network and application level monitoring as the results are similar.

Window-based Monitoring. The above experiments are performed under the instantaneous model. Figure 43 shows the performance of the estimation methods we introduced in Section 5.3 over the Planetlab trace under the window based model, where the x-axis marks the monitoring window size and the y-axis shows the ratio between the total number of sampling performed by one scheme and that performed by a periodical sampling scheme with the default sampling frequency. Here “instantaneous-0.01” refers to the estimation method for instantaneous violations with an error allowance $err = 0.01$, and “window-0.01” refers to the method for window based violations with $err = 0.01$. Recall that instantaneous detection reports state alerts whenever the monitored value exceeds a threshold while window based detection reports state alerts only when the monitored value continuously exceeds the threshold within a time window. As the instantaneous estimation method does not take window sizes into consideration, its corresponding cost reduction remains the same. The window based estimation method achieves more cost reduction as the window size increases (less likely to detect alerts), since larger window sizes lead to smaller violation likelihood which in turn allows the scheme to employ larger sampling intervals.

Distributed Sampling Coordination. Figure 44 illustrates the performance of different error allowance distribution schemes in network monitoring tasks. To vary the cost reduction yield on monitors, we change the local violation rates by varying the local thresholds. Initially, we assign a local threshold to each monitor so that all monitors have the same local violation rate. We then gradually change the local violation rate distribution to a Zipf distribution[129] which is commonly used to approximate skewed distributions in many situations. The x-axis shows the skewness of the distribution, and the distribution is uniform when skewness is 0. The y-axis shows the the ratio between the total number of sampling performed by one scheme and that performed by a periodical sampling scheme with the default sampling frequency (same as in Figure 43). We compare the performance of our iterative tuning scheme (adapt) described in Section 5.4 with an alternative scheme (even) which always divides the global error allowance evenly among monitors.

We see that the cost reduction of the even scheme gradually degrades with increasing skewness of the local violation rate distribution. This is because when the cost reduction yields on monitors are not the same, the even scheme cannot maximize the cost reduction yield over all monitors. The adaptive scheme reduces cost significantly more as it continuously allocates error allowance to monitors with high yields. Since a few monitors account for most local violations under a skewed Zipf distribution, the adaptive scheme can move error allowance from these monitors to those with higher cost reduction yield.

5.6 Related Work

Most existing works in distributed state monitoring [60, 57, 83] study the problem of employing distributed constraints to minimize communication cost of distributed state monitoring. Earlier works [60, 57] focus on the instantaneous model where a state alert is triggered whenever the sum of monitored values exceeds a threshold. Recent work[83] studies efficient distributed triggers for the more practical window based model[1]. While these works study the communication-efficient detection of state violations in a distributed manner based on the assumption that monitoring data is always available (with no cost), we study a lower level problem on collecting monitoring data. We investigate the fundamental relation between sampling intervals and accuracy, and propose violation likelihood based dynamic sampling schemes to enable efficient monitoring with controlled accuracy. In addition, while most of these works address only tasks using SUM aggregation, our approach can support different types of aggregation such as MAX and MIN, as long as the local threshold is set in a way to ensure monitoring correctness.

Previous distributed stream monitoring works [86, 88] study efficient algorithms for aggregating distributed data streams. Recent work by Jain et al.[55] proposes a set of basic metrics for measuring the accuracy of distributed monitoring results. Problems studied in these works are quite different from the ones we study in this chapter. While these works assume all stream data are pushed to the algorithm, Volley is designed for asynchronous monitoring where sampling operations are used to provide partial data (i.e. the information between two consecutive samples is not available). Hence, the sampling interval is an important factor for Volley, but is not as relevant to data streams.

A number of existing works in sensor networks use correlation to minimize energy consumption on sensor nodes[98, 29]. Our work differs from these works in several aspects. First, these works[98] often leverage the broadcast feature of sensor networks, while our system architecture is very different from sensor networks and does not have broadcast features. Second, we aim at reducing sampling cost while these works focus on reducing communication cost to preserve energy. Third, while sensor networks usually run a single or a few tasks, we have to consider multi-task correlation in large-scale distributed environments. Finally, some works (e.g. [29]) make assumptions on value distributions, while our approach makes no such assumptions.

Some scenarios such as network monitoring employ random sampling to collect a partial snapshot for state monitoring (e.g., a random subset of packets [42, 96]). Volley is complementary to random sampling as it can be used together with random sampling to offer additional cost savings by scheduling sampling operations. In addition, when a complete snapshot is required and random sampling is not suitable (e.g., random sampling may miss considerable packets related with a DDoS attack when the overall traffic volume is high, or a task may require a complete network snapshot to detect a certain interaction pattern), Volley can be used to provide monitoring efficiency and scalability.

CHAPTER VI

RELIABLE STATE MONITORING IN CLOUD DATACENTERS

6.1 Introduction

Dependable state monitoring is a fundamental building block for many distributed applications and services hosted in cloud datacenters. State monitoring is widely used to determine whether the aggregated state of a distributed application or service meets some predefined conditions[82]. Examples of state monitoring are prevalent. A web application owner may use state monitoring to check if the aggregated access observed at distributed application-hosting servers exceeds a pre-defined level[94]. State monitoring can also be used to detect DDoS attacks launched from platforms such as botnets[46].

Much existing state monitoring research efforts have been focused on minimizing the cost and the performance impact of state monitoring. For example, a good number of state monitoring techniques developed in this line of work focus on the threshold based state monitoring by carefully partitioning monitoring tasks between local nodes and coordinator nodes such that the overall communication cost is minimized [39, 94, 57, 79, 82]. Studies along this direction often make strong assumptions on the dynamic nature of the monitoring environments, such as unlimited bandwidth at participating monitoring nodes and 100% availability/responsiveness. However, such assumptions often do not hold in real deployment.

Unexpected performance anomalies and failures of computing nodes often introduce message delay and loss to monitoring related communications. Monitoring approaches designed without considering such messaging dynamics would inevitably produce unreliable results. Even worse, users are left in the dark without knowing that the monitoring output is no longer reliable. For instance, state monitoring approaches that rely on always-responsive

Table 1: Examples of State Monitoring

Applications	Description
Content Delivery	Monitoring the total access to a file mirrored at multiple serversto decide if serving capacity is sufficient.
Rate Limiting[94]	Limiting a user's total access towards a cloud service deployed at multiple physical locations.
Traffic Engineering[44]	Monitoring the overall traffic from an organization's sub-network (consists of distributed hosts) to the Internet.
Quality of Service[95]	Monitoring and Adjusting the total delay of a flow which is the sum of the actual delay in each router on its path.
Fighting DoS Attack	Detecting DoS Attack by counting SYN packets arriving at different hosts within a sub-network.
Botnet Detection[46]	Tracking the overall simultaneous TCP connections from a set of hosts to a given destination.

nodes may wait for messages from failed nodes indefinitely. Similarly, approaches that assume instantaneous message delivery may fail to report alerts on time when an important monitoring message is delayed. In both cases, users are not aware of potential errors in the monitoring results. Consequently, actions performed based on such unreliable results can be harmful or even catastrophic[55]. In addition, some recent works [109] proposed to rank monitoring alerts based on their false positive/negative rates. While this approach provides useful prioritizing of monitoring alerts generated by a collection of performance monitoring tasks, they do not consider monitoring error introduced by communication issues.

In this chapter, we present a new state monitoring framework that incorporates messaging dynamics in terms of message delay and message losses into monitoring results reporting and distributed monitoring coordination. Our framework provides two fundamental features for state monitoring. First, it estimates the accuracy of monitoring results based on the impact of messaging dynamics, which provides valuable information for users to decide whether monitoring results are trustworthy. Second, it minimizes the impact of dynamics whenever possible, which allows the state monitoring system to continuously adapt to changes in the system and to strive to produce reliable monitoring. These two features are important for large-scale distributed systems where dynamics and failures are the norm rather than the exception[36]. When combined, these two features shape a reliable state monitoring model that can tolerate communication dynamics and mitigate their impact on monitoring results.

To the best of our knowledge, our approach is the first state monitoring framework that explicitly handles messaging dynamics in large-scale distributed monitoring. We perform extensive experiments, including both trace-driven and real deployment ones, to evaluate our approach. The results suggest that our approach can produce good accuracy estimation and minimize monitoring errors introduced by messaging dynamics via adaptation. Furthermore, we also demonstrate its effectiveness by using our approach to support cloud application auto-scaling[4] which dynamically provisions new server instances when monitoring detects application workload bursts. Compared with existing techniques, our approach significantly reduces problematic monitoring results with the presence of messaging dynamics, and improves application response time by up to 30%.

6.2 Problem Definition

In this section, we briefly introduce existing state monitoring techniques, and reveal how messaging dynamics such as message delay and loss can impact the accuracy of monitoring results. We then outlier fundamental requirements for reliable state monitoring and challenges in meeting these requirements.

6.2.1 Preliminary

State monitoring is widely used for detecting anomalies in many distributed systems. For example, service providers usually monitor the overall request rate on a web application deployed over multiple hosts, as they want to receive a state alert when the overall request rate exceeds a threshold, e.g. the capacity limit of provisioned hosts. We refer to this type of monitoring as *state monitoring*, which continuously evaluates if a certain aspect of the distributed application, e.g. the overall request rate, deviates from a normal state. Table 1 summaries some of the popular state monitoring scenarios.

Most existing state monitoring studies employ an instantaneous state monitoring model, which triggers a state alert whenever a predefined threshold is violated. Specifically, the instantaneous state monitoring model[39, 86, 60, 97, 11, 57] detects state alerts by comparing the current aggregate value with a global threshold. Specifically, given the monitored value on monitor i at time t, $x_i(t), i \in [1, n]$, and the global threshold T, it considers the state at time t to be abnormal and triggers a state alert if $\sum_{i=1}^{n} x_i(t) > T$, which we refer to as *global violation*.

To perform instantaneous state monitoring, the line of existing work employs a distributed monitoring framework with multiple monitors and one coordinator (Figure 56). The global threshold T is decomposed into a set of local thresholds T_i for each monitor i such that $\sum_{i=1}^{n} T_i \leqslant T$. As a result, as long as $x_i(t) \leqslant T_i, \forall i \in [1, n]$, i.e. the monitored value at any node is lower or equal to its local threshold, the global threshold is satisfied because $\sum_{i=1}^{n} x_i(t) \leqslant \sum_{i=1}^{n} T_i \leqslant T$. Clearly, no communication is necessary in this case. When $x_i(t) > T_i$ on monitor i, it is possible that $\sum_{i=1}^{n} x_i(t) > T$ (global violation). In this case, monitor i sends a message to the coordinator to report *local violation* with the value $x_i(t)$. The coordinator, after receiving the local violation report, invokes a *global poll* procedure where it notifies other monitors to report their local values, and then determines whether $\sum_{i=1}^{n} x_i(t) \leqslant T$. The focus of existing work is to find optimal local threshold values that minimize the overall communication cost. For instance, if a monitor i often observes relatively higher x_i, it may be assigned with a higher T_i so that it does not frequently report local violations to the coordinator and trigger global polls.

6.2.2 Reliable State Monitoring and Challenges

Existing state monitoring work[39, 86, 60, 97, 11, 57, 82] often share the following assumptions: 1) nodes involved in a monitoring task is perfectly reliable in the sense that they are always online and responsive to monitoring requests; 2) a monitoring message can always be reliably and instantly delivered from one node to another. These two assumptions, however, do not always hold in cloud datacenter environments. First, cloud applications and services are often hosted by a massive number of distributed computing nodes. Failures, especially transient ones, are common for nodes of such large-scale distributed systems[36, 26]. Second, cloud datacenters often employ virtualization techniques to consolidate workloads and provide management flexibilities such as virtual machine cloning and live migration. Despite its benefits, virtualization also introduces a number of challenges such as performance interferences among virtual machines running on the same physical host. Such interferences could introduce serious network performance degradation, including heavy message delays and message drops[47, 93].

To provide robustness against messaging dynamics, recent work proposed by Jain et al.[55] employs a set of coarse network performance metrics that are continuously updated to reflex the status of monitoring communication. One example of such metrics is the number of nodes that contributed to a monitoring task. The intention of providing these metrics is to allow users to decide how trustworthy monitoring results are based on values of these metrics. While this approach certainly has its metrics in certain monitoring scenarios, it has a number limitations.

First, it considers the status of a monitor as either online or offline, and overlooks situations where message delays also play an important role. For instance, a monitor node may appear online, but it may introduce considerable latencies to messages sent to or received from it. Such message delays are as important as message loss caused by offline nodes, because they may also lead to mis-detection of anomalies. In fact, anecdotal evidences[9] suggest that communication latency in virtualized cloud datacenters can be a serious issue.

Second, and more importantly, it is difficult for users to interpret the impact of reported network level issues on monitoring accuracy. If one of the nodes fails to report its local monitoring data, does the corresponding monitoring result still reliable? The problem gets aggravated in large-scale distributed monitoring where message delay or loss can be quite common given the number of participating nodes, e.g. hundreds of web servers for large cloud applications, and even thousands of servers for Hadoop clusters. On the one hand, if we simply invalidate the monitoring results whenever message delay or loss occurs, we

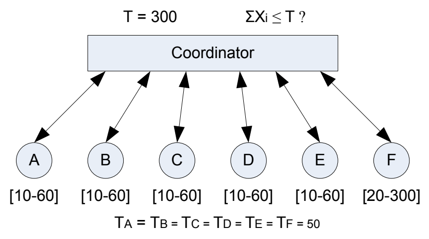


Figure 46: A Motivating Example

would end up with frequent gaps in monitoring data and low monitoring utility. On the other hand, if we choose to use such monitoring results, how should we perform *accuracy estimation* for monitoring results given the observed message delay and loss?

Figure 56 shows a motivating example where a monitoring task involves one coordinator and six monitors (A to F). The monitoring goal is to check if the aggregated request rates observed on each monitor (x_i) exceeds the global threshold $T = 300$. The numbers under each monitor indicate the range of values observed by the monitor. Such range statistics can be obtained through long-term observations. For simplicity, we assume local thresholds employed by all monitors have the same value 50, i.e. $T_A = T_B = T_C = T_D = T_E = T_F = 50$.

Estimating monitoring accuracy based on messaging dynamics information is difficult. Simply using the scope of message delay or loss to infer accuracy can be misleading. For example, if monitor A, B, and C (50% of total monitors) all fail to response in a global poll during a transient failure, one may come to the conclusion that the result of global poll should be invalidated as half of the monitors do not contribute to the result. However, as monitor A, B and C observe relatively small monitored values, the corresponding global poll results may still be useful. For instance, if the global poll suggests that $x_D + x_E + x_F = 100$, we can conclude that there is no global violation, i.e. $\sum_i^{i=\{A...F\}} x_i \leqslant 300$ with high confidence, because the probability of $\sum_i^{i=\{A...C\}} x_i \leqslant 180$ is fairly high given observed value ranges of A, B and C. On the contrary, if monitor F fails to response, even though F is only one node, the uncertainty of monitoring results increases significantly. For example, if $\sum_i^{i=\{A...E\}} x_i = 150$, it is hard to tell whether a global violation exists due to the high variance of F's observed values.

An ideal approach should provide users an intuitive accuracy estimation such as "the current monitoring result is correct with a probability of 0.93", instead of simply reporting the statistics of message delay or loss. Such an approach must address the challenge in quantitatively estimating the accuracy of monitoring results based on messaging quality information. It should be aware of state monitoring algorithm context as the algorithm has two phases, the local violation reporting phase and global poll phase. It should also utilize information on both messaging quality and per-monitor value distributions to offer the best estimation possible.

Third, accuracy estimation alone is not enough to provide reliable monitoring and minimize the impact of messaging quality degradation. Resolving node failures may take time. Network performance degradation caused by virtual machine interferences often lasts for a while until one virtual machine is migrated to other hosts. As a result, messaging dynamics can last for some time. Without monitoring self-adaptation and minimizing the corresponding accuracy loss, users may lose access to any meaningful monitoring result during a fairly long period, This is clearly not acceptable to cloud users, especially for those who pay for using commercial cloud monitoring services such as CloudWatch[1]. For instance, if node F continuously experiences message loss, local violation reports sent from F are very likely to be dropped. Consequently, the coordinator does not trigger global polls when it receives no local violation reports. If a true violation exists, e.g. $x_A = 45, x_B = 45, x_C = 45, x_D = 45, x_E = 45, x_F = 110$ and $\sum_i^{i=\{A...F\}} x_i = 335$, the coordinator will mis-detect it.

One possible approach to reduce monitoring errors introduced by such messaging dynamics is to let healthy nodes, i.e. nodes not affected by messaging dynamics, to report their local values at a finer granularity to compensate the information loss on problem nodes. In the above example, if we reduce local thresholds on node A, B, C, D, E to 30. the coordinator will receive local violations from node A, B, C, D and E, and trigger a global poll. Even if F also fails to response to the global poll, the coordinator can find that $\sum_i^{i\in\{A,\ldots,E\}} x_i = 225$. For the sake of the example, suppose x_F is uniformly distributed over $[20, 300]$. The coordinator can infer that the probability of a global violation is high. This is because a global violation exists if $x_F > 75$ which is very likely (> 0.8) given x_F's distribution. Similarly, adaptation can also be used to rule out the possibility of global violations. For instance, if node E is troubled by messaging dynamics, instead of worrying whether E would cause coordinator fail to trigger global polls, we can increase E's local threshold to 70 so that the probability of detecting local violation on E is trivial. Correspondingly, we also reduce the thresholds on the rest of the nodes to 45 to ensure the correctness of monitoring ($\sum_i T_i \leqslant T$). As a result, as long as $\sum_i^{i\in\{A,\ldots,D,F\}} x_i < 230$, we can infer that there is no global violation with high probability, even though node E is under the impact of messaging dynamics.

While this type of self-adaptation seems promising, designing such a scheme is difficult and relies on answers to a number of fundamental questions: how much should we reduce the local threshold on E? For a more general case, how should we divide the global thresholds when there are multiple problem nodes to minimize the possible error they may introduce, especially when they observes different levels of message and delay? In the rest of this chapter, we address these challenges in accuracy estimation and self-adaptation and present details of our reliable state monitoring approach.

6.3 Reliable State Monitoring

State monitoring continuously checks whether a monitored system enters a critical pre-defined state. Hence, state monitoring tasks usually generate binary results which indicate either "state violation exists"(positive detection) or "no state violation exists"(negative detection). Beyond this basic result, our reliable state monitoring approach also marks the estimated accuracy of a monitoring result in the form of error probabilities. For positive detections, the error probability is the probability of false positives. The error probability is the probability of false negatives for negative detections.

To perform accuracy estimation, we design estimation schemes for both local violation reporting and global poll processes respectively. These schemes leverage the information on messaging dynamics and per-node monitored value distributions to capture uncertainties caused by messaging dynamics. In addition, we also examine the unique problem of out-of-order global polls caused by message delay. The final accuracy estimation results synthesize the uncertainties observed at different stages of the state monitoring algorithm.

Besides accuracy estimation, our approach also minimizes errors caused by non-transient messaging dynamics via two parallel directions of adjustments on distributed monitoring parameters. One tries to minimize the chance that troubled nodes deliver local violation reports to the coordinator. Since they may fail to deliver local violation reports, such adjustments essentially minimizes the uncertainties caused by them. The other direction of adjustments is to configure healthy nodes to report their local monitored values more often. This allows the coordinator to make better accuracy estimation which in turn helps to detect or rule out a global violation with high confidence, e.g. "no global violation exists" with an estimated false negative probability of 0.005 is very likely to be true.

6.3.1 Messaging Dynamics

Although a cloud datacenter may encounter countless types of failures and anomalies at different levels (network/server/OS/etc.), their impact on monitoring related communication can often be characterized by message delay and message loss. For brevity, we use the term *messaging dynamics* to refer to both message delay and loss. Depending on the seriousness of messaging dynamics, the monitoring system may observe different difficulties in inter-node communication, from slight message delay to complete node failure (100% message loss rate or indefinite delay).

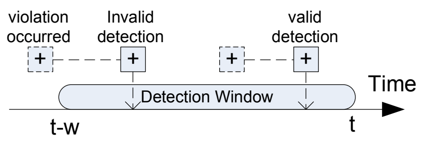


Figure 47: Detection Window

The focus of our study is utilizing message delay and loss information to provide reliable state monitoring functionalities via accuracy estimation and accuracy-driven self-adaptation. Our approach obtains message delay and loss information in two ways. One is direct observation in global polls, e.g. the coordinator knows whether it has received a response from a certain monitor on time. The other is utilizing existing techniques such as [55] to collect pair-wise message delay and loss information between a monitor and the coordinator. Note that our approach is orthogonal to the messaging quality measurement techniques, as it takes the output of the measurement to perform accuracy estimation and self-adaptation. Our approach only requires basic messaging dynamics information. For message delay, it requires a histogram that records the distribution of observed message delays. For message loss, it takes the message loss rate as input.

6.3.2 Detection Window

We introduce the concept of detection window to allow users to define their tolerance level of result delays. Specifically, a detection window is a sliding time window with length w. We consider a global violation V detected at time t a correctly detected one if its actual occurrence time $t_o \in [t - w, t]$. Note that multiple global violations may occur between the current time t and $t - w$ as Figure 47 shows. We do not distinguish different global violations within the current detection window, as users often care about whether there exists a global violation within the detection window instead of exactly how many global violations are there. The concept of detection window is important for capturing the dynamic nature of state monitoring in real world deployment.

6.3.3 Accuracy Estimation

Recall that the distributed state monitoring algorithm we introduced in Section 6.2.1 has two stages, the local violation reporting stage and the global poll stage. As message delay and loss have impact on both stages, our analysis on their accuracy impact needs to be conducted separately. We next study the problem of accuracy estimation for the original state monitoring algorithm by looking into the two stages separately.

When message delay or loss occurs during local violation reporting, the coordinator may fail to receive a local violation report and trigger a global poll in time. Consequently, it may mis-detect a global violation if one does exist, and introduce false negative results. To estimate the monitoring accuracy at this stage, the coordinator continuously updates the estimated probability of failing to receive one or more local violations based on the current messaging dynamics situation and per-monitor value distribution. When message delay or loss occurs during a global poll, the coordinator can not collect all necessary information on time, which again may cause the coordinator to mis-detect global violation and introduces false negatives. Hence, we estimate the probability of mis-detecting a global violation based on collected values during the global poll and the value distribution of troubled monitors.

Local Violation Reporting. To facilitate the accuracy estimation at the local violation reporting stage, each monitor maintains a local histogram that records the distribution of local monitored values. Much previous research[86, 97, 54, 57, 82] suggests that such distribution statistics of recent monitored values provides good estimation on future values. Specifically, each monitor maintains a histogram of the values that it sees over time as $H_i(x)$ where $H_i(x)$ is the probability of monitor i taking the value x. We use equi-depth histograms to keep track of the data distribution. For generality purposes, we assume that the monitored value distribution is independent of messaging dynamics. To ensure that the histogram reflects recently seen values more prominently than older ones, each monitor continuously updates its histogram with exponential aging. A monitor also periodically sends its local histogram to the coordinator.

We first look at the probability of monitor i fails to report a local violation which can be computed as follows,

$$P(f_i) = P(v_i)P(m_i)$$

, where $P(v_i)$ is the probability of detecting a local violation on monitor i, and $P(m_i)$ is the probability of a message sent from monitor i failing to reach the coordinator due to messaging dynamics. $P(v_i) = P(x_i > T_i)$ where x_i and T_i are the monitored value and the local threshold on monitor i respectively. $P(x_i > T_i)$ can be easily computed based on T_i and the distribution of x_i provided by the histogram of monitor i. $P(m_i)$ depends on the situation of message delay and loss. Let $P(p_i)$ be the probability of a message sent from monitor i to the coordinator being dropped. Let $P(d_i)$ be the probability of a reporting message sent from monitor i to the coordinator being delayed beyond users' tolerance, i.e. the local violation report is delayed more than a time length of w (the detection window size) so that the potential global violation associated with the delayed local violation report becomes invalid even if detected. Given $P(p_i)$ and $P(d_i)$, we have

$$P(m_i) = 1 - (1 - P(p_i))(1 - P(d_i))$$

The rational here is that for a local violation report to successfully reach the coordinator, it must not being dropped or heavily delayed at the same time. Both $P(p_i)$ and $P(d_i)$ can be easily determined based on the measurement output of messaging dynamics. $P(p_i)$ is simply the message loss rate. $P(d_i)$ can be computed as $P(d_i) = P(l_i > w)$, where l_i is the latency of messages sent from monitor i to the coordinator, and $P(l_i > w)$ is easy to obtain given the latency distribution of messages. Clearly, $P(m_i)$ grows with $P(p_i)$ and $P(d_i)$ and $P(m_i) = 0$ when messaging dynamics do not exist.

During the local violation reporting phase, the overall probability of the coordinator failing to receive local violations $P(F)$ depends on all monitors. Therefore, we have

$$P(F) = 1 - \prod_{i}^{n}(1 - P(f_i))$$

, where n is the number of monitors and we consider local violations on different monitors are independent for generality. Clearly, $P(F)$ grows with the number of problem monitors, i.e. monitors experiencing delay or loss. With $P(F)$, the probability of false negatives caused by missing local violation reports P_l can be estimated as $P_l = cP(F)$ where c is referred as the conversion rate between local violations and global violations. The coordinator maintains c based on its observations on previous local violations and global violations.

Global Polls. Recall that in the original state monitoring algorithm, when the coordinator receives a local violation report, it initiates the global poll process, where it requests all monitors to report their *current* local monitored values. However, when message delay and loss exist, the coordinator may receive a delayed report about a local violation that actually occurs at an earlier time t. As a result, when the coordinator invokes a global poll, it requests all monitors to report their *previous* local monitored values observed at time t. To support this functionality, monitors locally keep a record of previous monitored values observed within a sliding window with size w. Monitored values observed even earlier are not required to keep as those are invalid if they are associated with a global poll.

Once the coordinator initiates the global poll process, our accuracy estimation also enters the second stage, where we estimate the possibility of mis-detecting global violations due to message delay and loss in the global poll process. The estimation starts when the coordinator does not receive all responses on time. Since the coordinator does not report anything until it receives all monitoring data, the probability of detecting a state violation given the set of received monitored values is

$$P(V) = P\{\sum_{i \in K} x_i > T - \sum_{i \in \bar{K}} x_i\} \tag{17}$$

, where K is the set of monitors whose responses do not reach the coordinator, and $\bar{K}$ are the rest of the monitors. The right hand side of the equation can be determined based on the value histogram of monitors. At any time point, the probability of detecting global

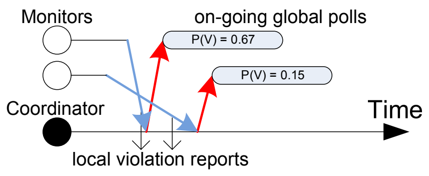


Figure 48: Out-of-order Global Polls

violation is the probability of detecting global violation within the time window of delay tolerance.

Out-of-Order Global Polls. Due to the existence of delay, local violation reports sent from different monitors may arrive out-of-order. Accordingly, as new global poll processes may start before previous global poll processes finish, the coordinator may be involved in multiple ongoing global poll processes at the same time as Figure 48 shows.

When the coordinator receives local violation reports r, it first checks its timestamp t_r (local violation occurring time) to see if $t_r \geqslant t - w$ where t is the current time (report receiving time) and w is the user-specified detection window size. If true, it ignores the local violation report as the violation report is expired. Otherwise, it initiates a global poll process and use t_r as its timestamp. As each global poll may take different time to finish (due to message delay or loss), the coordinator continuously checks the lifetime of global polls and removes those with t_r that $t_r \geqslant t - w$.

For accuracy estimation, users are interested in whether there exists one or more global violations within the time interval of $[t - w, t]$. When there are multiple ongoing global polls, it means that there are multiple potential global violations requiring verification. Accordingly, our accuracy estimation should be on whether there exists at least one ongoing global poll leading to global violation.

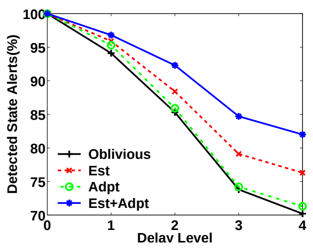
Let $P_j(V)$ be the probability of triggering global violation in global poll j. $P_j(V)$ can be determined based on Equation 17. The probability P_g of at least one global poll out of M ongoing ones triggering global violation is

$$P_g = 1 - \Pi_{j=1}^{M}(1 - P_j(V))$$

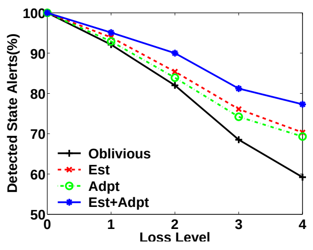
Clearly, P_g increases quickly when the coordinator observes growing number of ongoing global polls. If P_g is sufficiently high, our monitoring algorithm will report possible state violation. This is particularly useful for situations with a few monitors suffering serious message delay or loss, because no global polls can finish if these nodes can not send their responses in time and the coordinator can never trigger global violation if running existing state monitoring algorithms.

Combining Estimations of Both Stages. While we have considered the accuracy estimation problem for local violation reporting and global poll stages separately, a running coordinator often experiences both local violation failures and incomplete global polls at the same time. Hence, combining estimation on both stages is critical for delivering correct accuracy estimation results. The overall probability of false negatives can be computed as $\beta = 1 - (1 - P_l)(1 - P_g)$ where P_l and P_g are the probability of false negatives introduced by failed local violation reporting and global polls respectively. Note that $\beta \neq P_l + P_g$ as the event of miss-detecting a global violation due to failed local violation reporting, and the event of miss-detecting a global violation due to failed global polls are not mutually exclusive.

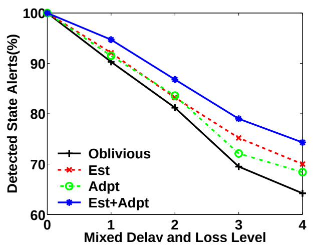
A Balanced State Monitoring Algorithm. The original state monitoring algorithm invokes global polls only when receives local violation reports, and triggers state alerts only after the coordinator collects responses from all monitors. When messaging dynamics exist, such a deterministic algorithm has two issues. First, it may miss opportunities to invoke global polls. Second, it never produces false positive results, but may introduce many false negatives results. We introduce a balanced state monitoring algorithm that minimizes the overall monitoring error. The balanced algorithm is obtained through two revision on the original algorithm. First, when $P(F)$, the probability of failing to receive local violation reports at the coordinator, is sufficiently large (e.g. $\geqslant 0.95$), the algorithm triggers a global poll. Second, if the estimated false negative probability β in the global poll phase raises above 50%, the monitoring algorithm also reports state violation with a false



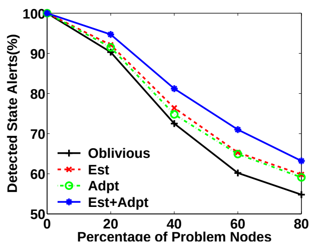
(a)



(b)



(c)



(d)

Figure 49: State Violation Detection Rate: (a)under increasing level of delay; (b) under increasing level of message loss;(c)under increasing level of mixed message delay and loss; (d) with increasing number of problem monitors

positive probability $1 - \beta$. The balanced algorithm is more likely detect global violations compared with the original algorithm, especially when β is large.

6.3.4 Accuracy-Oriented Adaptation

Sometimes monitors may experience long-lasting message loss and delays. For instance, a Xen-based virtual machine continuously generating intensive network IO may cause considerable workload on domain 0, which further leads to constant packet queueing for other virtual machines running on the same host[47, 93]. As a result, monitor processes running on troubled virtual machines would experience continuous messaging dynamics until the performance interference is resolved. For these situations, we argue that providing accuracy estimation alone is not enough. Reliable state monitoring should also adapt to such non-transient messaging dynamics and minimize accuracy loss whenever possible.

The general idea of our approach is to minimize the error introduced by local violation reporting through proper adjustment of local thresholds. We know that the distributed state monitoring algorithm employs local thresholds to minimize the amount of local violation reports sending to the coordinator. However, this technique introduces extra uncertainties when messaging dynamics exist, because the coordinator cannot distinguish the case where a monitor does not detect local violation from the case where a monitor fails to report a local violation. Our approach minimizes such uncertainties through two simultaneous adjustments. On one hand, it adjusts local thresholds on troubled monitors to reduce its chance of detecting local violations, as the corresponding reports may not arrive the coordinator which in turn introduces uncertainties. On the other hand, it also adjusts local thresholds on healthy monitors to increase their local violation reporting frequencies to maximize the information available to the coordinator so that it can provide good accuracy estimation. The adjustment on healthy monitors is also important for monitoring correctness where we ensure $\sum_i^n T_i \leqslant T$.

Our motivating example in Section 6.2.2 shows two simple examples where a single node (F/E) experiences messaging dynamics and adjustment of local thresholds leads to confirm or rule out a global violation. Situations in real world monitoring environments, however, are more complicated, e.g. multiple troubled monitors experiencing different levels of messaging dynamics. As the impact of message delay and loss to local violation reporting can be measured by the expected number of failed local violation reports $E(\text{fr})$, we formulate the local threshold adjustment problem as a constrained optimization problem as follows,

$$\begin{aligned} \min \quad & E(fr) = \Sigma_i^n P(v_i|T_i)P(m_i) \\ \text{s.t.} \quad & \Sigma_i^n T_i \leqslant T \end{aligned}$$

, where $P(v_i|T_i)$ is the conditional probability of reporting local violation on monitor i given its local threshold T_i and $P(m_i)$ is the probability of failing to send a message to the coordinator. One difficulty here is that we do not have a closed form for $P(v_i|T_i) = P(x_i > T_i)$ because we approximate the distribution of x_i with histograms. While one could use a brutal force approach to try all possible combinations of T_i, such an approach may not scale well for large-scale distributed monitoring tasks with hundreds or even thousands of monitors. To address this issue, we replace $P(v_i|T_i)$ with its upper bound $\overline{P(v_i|T_i)}$ by applying Markov's inequality (Chebyshev's inequality does not yield a closed form) where $P(|x_i| > T_i) \leqslant \frac{E(|x_i|)}{T_i}$. Since x_i is positive in most scenarios and $E(|x_i|)$ can be obtained through x_i's histograms, applying this approximation and Lagrange multiplier leads us to a closed form solution. One remaining issue is that the above optimization often sets T_i too high for troubled monitors due to the limited tightness of Markov bound. Hence, after we obtain a solution, we limit T_i to $\mu + 3\sigma_i$ where σ_i is the standard deviation of x_i. Note that healthy monitors are not involved in the above optimization. When $T_{res} = T - \sum_i^{i \in K} T_i > 0$ where K is the set of troubled monitors, we set $T_i = T_{res}/|\overline{K}|$, where $\overline{K}$ is the set of healthy nodes, to maintain local violation reporting efficiency whenever possible. We find the resulting adjustments perform well in practice. In addition, we invoke adaptation only when at least one node experiences relatively long-lasting (e.g. 5 minutes) messaging dynamics to avoid frequent adaptation.

6.3.5 Discussion

Large-scale state monitoring tasks often employ hierarchical topologies where monitors are organized into multi-level trees[54]. We developed techniques for supporting accuracy estimation and self-adaptation in such hierarchical monitoring trees. In addition, certain state monitoring tasks rely on advanced state monitoring models such as window based models[82] to handel noisy data and outliers, we also devised schemes to support such advanced state monitoring models. Due to space limitations, we leave details of these techniques to our technical report[80].

We present our approach based on monitoring tasks defined over sum-aggregation and upper bound threshold for brevity. Our approach also supports other types of tasks, such as global violations defined by lower bound threshold, i.e. global violations triggered by $\sum_i^n x_i < T_i$, and aggregations formed by linear combinations of x_i, e.g. global violations triggered by $AVG(x_i) > T_i$. The required modification to our approach is trivial for these tasks[80].

6.4 Evaluation

We implemented a prototype monitoring system with our reliable monitoring approach and evaluated its performance with both monitoring traces collected from real systems and cloud applications. We highlight key observations in our experiment as follows.

- Our reliable state monitoring approach detects up to 20% more state violations under considerable messaging dynamics, compared with existing state monitoring approaches.

- By applying our techniques to cloud application auto-scaling, we can reduce application response time and the number of timeout requests by up to 30% when monitoring related messages are subject to delay and loss.

6.4.1 Experiment Setup

Our experiments consist of both trace-driven simulation and real system evaluation. The trace-driven experiment evaluates the performance of our approach with access traces of WorldCup 1998 official website hosted by 30 servers distributed across the globe[18]. We used the server log data consisting of 57 million page requests distributed across servers. We evaluate the monitoring accuracy achieved by our approach for a variety of messaging dynamics in this set of experiments. The other part of our experiments leverages our monitoring techniques to support auto-scaling of cloud applications where server instances can be added to the resource pool of an application dynamically based on current workload[4]. We deploy a distributed RUBiS[8], an auction web application modeled after eBay.com for performance benchmarking, and use state monitoring to trigger new server instance provisioning. For the real system evaluation, we are interested in the impact of improved monitoring accuracy on real world application performance. We provide more details of experiment setup together with results in the rest of this section.

6.4.2 Results

Figure 49 shows the state violation detection percentage of different monitoring approaches under different level and types messaging quality degradation. Here the y-axis is the percentage of state violation detected by the monitoring algorithm over state violation detected by an oracle which can detect all violations in a given trace. In our comparison, we consider four monitoring algorithms: 1) Oblivious, the existing instantaneous monitoring algorithm which is oblivious to inter-node messaging quality; 2) Est, the instantaneous monitoring algorithm enhanced with our accuracy estimation techniques; 3) Adpt, the instantaneous monitoring algorithm enhanced with our accuracy-oriented adaptation techniques; 4) Est+Adpt, the instantaneous monitoring algorithm enhanced with both estimation and adaptation techniques.

We emulate a distributed rate limiting (Table 1) scenario and use a monitoring task that triggers state violations whenever it detects the overall request rate (the sum of request rate on all monitors) exceeds a global threshold (set to 3000 per second). The task involves 30 monitors, each of which monitors the request rate of one server by reading the corresponding server request trace periodically. Furthermore, we set the detection window size to be 15 seconds, which means a state violation is considered as successfully detected if the time of detection is at most 15 seconds later than the occurrence time of the state violation.

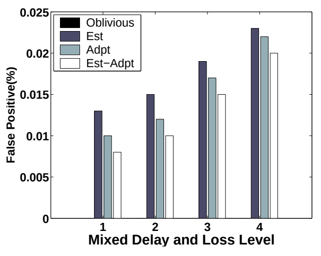
Figure 49(a) illustrates the performance of different algorithms under increasing message delay. Here the x-axis shows the levels of injected message delay. For delay on level $k (k = 0, 1, 2, 3, 4)$, we pick $20 \times k\%$ of messages of a problem monitor and inject a delay time randomly chosen from 5 to 60 seconds. By default, we randomly pick 10% of monitors to be problem monitors. While there are many ways to inject message delays, we use the above injection method for the sake of simplicity and interpretation. The detection rate of the oblivious algorithm drops quickly as delay level increases, primarily because its global poll process always waits until messages from all monitors arrive and the resulting delay on the violation reporting often exceeds the delay tolerance interval. The Est algorithm performs much better as it is able to estimate the probability of a state violation based on incomplete global poll results, which allows the Est scheme to report state violation when the estimated probability is high (above 0.9 in our experiment). For instance, when an incomplete global poll yields a total request rate close to the global threshold, it is very likely that a state violation exists even though responses from problem monitors are not available. The Adpt scheme, however, provides limited improvement when used alone. This is because accuracy-oriented adaptation by itself only reduces the chance of a problem monitor reporting local violation. Without accuracy estimation, the Adpt scheme still waits for all responses in global polls. With both accuracy estimation and adaptation, the Est+Adpt scheme achieves significantly higher detection rate.

In Figure 49(b), we use different levels of message loss to evaluate the performance of different algorithms. Similar to the injection of delay, we randomly pick $20 \times k\%$ message of a problem node to drop for a k-level message loss. The relative performance of the four algorithms is similar to what we observed in Figure 49(a), although the detection rate achieved by each algorithm drops slightly compared with that in Figure 49(a) as delayed messages often still help to detect state violation compared with completely dropped messages.

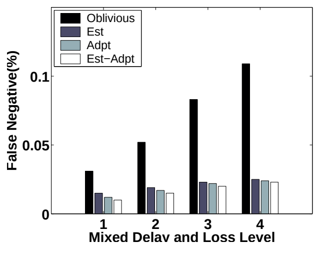
For the rest of the evaluation section, we inject mixed message delay loss, instead of mess delay or loss alone, for comprehensive reliability evaluation. Similarly, the k level delay and loss means that 10% messages are randomly chosen to drop and another 10% messages are randomly chosen to add delays. Figure 49(c) shows the violation detection performance of different algorithms given increasing levels of mixed message delay and loss. We observe similar results in this figure and the performance achieved by our approach lies between those achieved in the two previous figures. In Figure 49(d), we vary the scope of problem nodes from 20%(the default case) to 80%. The result suggests that our approach consistently improves monitoring accuracy. Nevertheless, when problem monitors becomes dominate, its performance is relatively worse that that in the three previous figures.

Figure 50(a) shows the corresponding percentage of false positive (reporting state violation when none exist) produced. Recall that the default instantaneous monitoring algorithm does not produce false positive as its global poll reports state violation only when the completely collected responses confirms the state violation, which, however, causes high false negative rate (shown in Figure 50(b)). Figure 50(b) shows the false negative (reporting no state violations when at least one exists) rates of all schemes. We can see that all of our three schemes achieve fairly low false positive and false negative rates.

Figure 51 illustrates the three key efforts our approach makes to improve monitoring accuracy and the corresponding portion of correctly reported state violations that are missed by the default instantaneous monitoring algorithm. Here *Adaptation* refers to the effort of

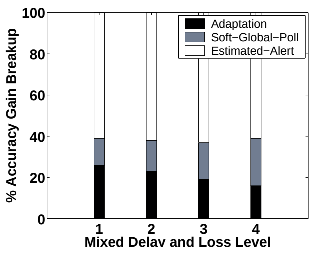


(a)

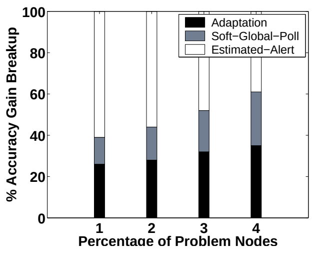


(b)

Figure 50: Errors in State Violation Detection: (a) comparison of false positive; (b) comparison of false negative;

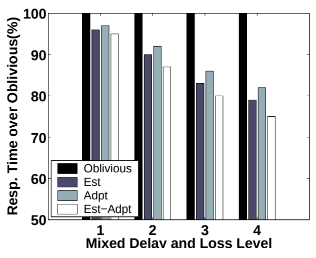


(a)

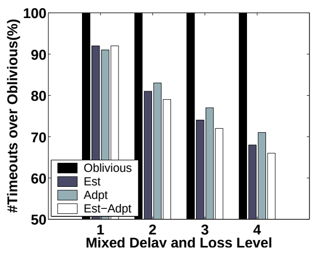


(b)

Figure 51: Accuracy Improvement Breakup: (a) with increasing message loss and delay levels; (b) with increasing percentage of problem monitors.



(a)



(b)

Figure 52: Impact on cloud application auto-scaling: (a) comparison of response time; (b) comparison of timeouts.

reconfiguring local threshold, *Soft-Global-Poll* refers to the effort of triggering global polls when the estimated local violation reporting probability is high (instead of receiving a local violation), and *Estimated-Alert* refers to the effort of reporting state violation when the estimated probability is sufficiently high. Note that multiple efforts may contribute to a correctly reported state violation at the same time. Among the three efforts in both Figure 51(a) and Figure 51(b), *Estimated-Alert* clearly contributes the most as incomplete global polls are the main reason for false negatives in the default monitoring algorithm.

Figure 52 shows the performance difference of RUBiS with auto-scaling enabled by different monitoring schemes. We deploy a PHP version of RUBiS in Emulab[120] where it has a set of web servers and a database backend. Each web server runs in a small footprint XEN-based virtual machine (1 vCPU) and the database runs on a dedicated physical machine. This is to ensure database is not the performance bottleneck. To run the experiment, we periodically introduce workload bursts to RUBiS's default workload, and use state monitoring to check if the total number of timeout requests on all web servers exceeds a given threshold, i.e. one monitor runs on one web server to monitor local timeout requests. RUBiS initially runs with 5 web servers. When violations are detected, we gradually add new web servers one by one to balance workloads until no violation is detected (auto-scaling). Similarly, when no violations are detected for 1 minute, we gradually remove dynamically added web servers one by one.

We introduce messaging delay and loss to monitor-coordinator communication in the same way as that in the trace-driven experiments. The y-axis of Figure 52 shows the average response time and timeout request number of RUBiS requests which are normalized by those of the oblivious scheme. Clearly, as our enhanced schemes detect more state violations, they can more reliability trigger auto-scaling when there is a workload burst, which in turn reduces response time and request timeout by up to 30%. In addition, accuracy estimation achieves higher detection rate compared with self-adaptation does. This is because monitors on load balanced web servers often observe similar timeouts and accuracy estimation can often confirm global violations based on partial monitoring data.

6.5 Related Work

Most existing state monitoring works[39, 97, 60, 11, 57] study communication efficient detection of constraint violation. This line of works adopt an instantaneous monitoring model where a state alert is triggered whenever the sum of monitored values exceeds a threshold. Meng et al.[] study a window-based state monitoring approach which captures only continuous threshold-violation to provide robustness against transient value outliers. Similarly, Huang et al.[49] propose a cumulative trigger to track the cumulative amount of value "overflows". These works do not consider the impact of messaging dynamics, and thus, may deliver unreliable monitoring results under churn.

Jain and et al.[55] studies the impact of hierarchical aggregation, arithmetic filtering and temporary batching in an unreliable network. They propose to gauge the degree of inaccuracy based on the number of unreachable monitoring nodes and the number of duplicated monitoring messages caused by DHT overlay maintenance. While this work provides insight for understanding the interplay between monitoring efficiency and accuracy given message losses, it also has several limitations as we mentioned in Section 6.2.2 such as not considering delay and difficulties in gauging monitoring accuracy. Our work is complementary to [55] as we try to move forward the understanding of monitoring reliability by studying accuracy estimation and self-adaptation in state monitoring.

Viswanathan and et. al. [109] recently proposed ranking windows of monitored metrics based on their probability of occurrence computed based on the false positive/negative rates for monitoring application performance anomalies. While this approach provides useful prioritizing of monitoring alerts generated by a collection of performance monitoring tasks, especially when administrators are overwhelmed by a large number of monitoring alerts, our approach aims at improving the monitoring accuracy of each individual monitoring task with the existence of data noises or communication issues.

CHAPTER VII

AN EFFICIENT PREDICTION-BASED MULTI-TIER CLOUD APPLICATION PROVISIONING PLANNING METHOD

7.1 Introduction

Deploying a multi-tier web application to meet a certain performance goal with minimum virtual instance renting cost is often the goal of many Infrastructure-as-a-Service (IaaS) users. It is, however, difficult to achieve due to several reasons. First, a typical IaaS (e.g., Amazon's EC2 and IBM's SCE) offers a variety of virtual server instances with different performance capacities and rental rates. Such instances are often marked with a high level description of their hardware/software configuration (e.g. 1 or 2 virtual CPU) which offers little information with regarding their performance for a particular application.

Second, multi-tier web applications often leverage clusters at different tiers to offer features such as load balance, scalability and fault tolerance. The configuration of clusters (e.g., the number of member nodes, how workloads are distributed among member nodes) has a direct impact on application performance. However, the relation between cluster configuration and performance is application-dependent, and often not clear to Cloud users.

To meet a given performance goal, users often over-provision a multi-tier web application by renting high-end virtual server instances and employing large clusters. Over-provisioning introduce high instance renting cost, which may make cloud deployment a less desirable option compared with traditional deployment options. Unfortunately, manually experimenting with different provisioning plans is often impractical given the huge space of candidate provisioning plans.

We propose a prediction-based provisioning planning method which can find the most cost-effective provisioning plan for a given performance goal by searching the space of candidate plans with performance prediction. This invention employs a set of novel techniques that can efficiently learn performance traits of applications, virtual machines and clusters to build models to predict the performance for an arbitrary provisioning plan. It utilizes historical performance monitoring data and data collected from a small set of automatic experiments to build a composite performance prediction model that takes application workloads, types of virtual server instances and cluster configuration as input, and outputs predicted performance.

Figure 53 shows the overall flow of the provisioning method. The proposed method avoids exhaustively performing experiments on all candidate deployments to build a performance prediction model by using a two-step performance prediction procedure. Instead of directly predicting the performance of an arbitrary deployment (target), it first predicts the performance on a known deployment (base) and then predicts the performance differences between the target deployment and the base deployment. It combines the predicted base performance and the predicted performance changes to obtain the performance on the target deployment. To achieve efficiency, the procedure predicts the performance change based on the deployment difference between the base deployment and the target deployment *within each tier*, rather than predicts the overall performance changes holistically cross multiple tiers. This avoids the need of exhaustively explores all deployments that represent combinations of deployment changes cross tiers, because it considers each tier independently. For instance, suppose we have an application consisting of 3 tiers and each tier has 10 possible forms. Exhaustive search would explore all $10^3 = 1000$ deployments to train a traditional performance prediction model, while our method only needs to test $3 * 10 = 30$ deployments to obtain our two-step performance prediction model. Our method also applies a multiplicative-delta learning technique (in capturing performance changes introduced by different sizes of a tier) to further reduce the number of required experiments for model training. In addition, our method includes techniques addressing cross-tier workload characteristics changes that violates the inter-tier independence of our

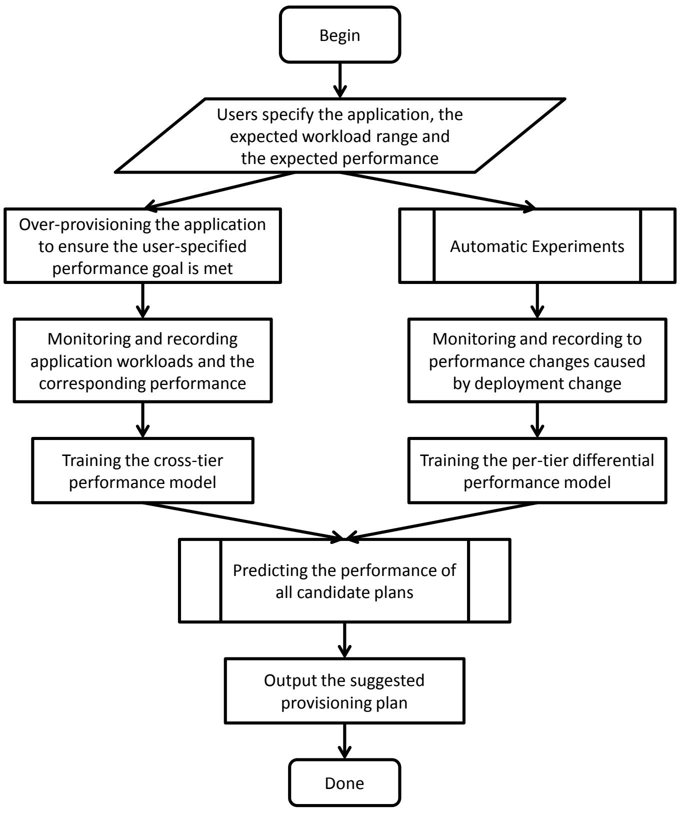

Begin
Users specify the application, the expected workload range and the expected performance
Over-provisioning the application to ensure the user-specified performance goal is met
Automatic Experiments
Monitoring and recording application workloads and the corresponding performance
Monitoring and recording to performance changes caused by deployment change
Training the cross-tier performance model
Training the per-tier differential performance model
Predicting the performance of all candidate plans
Output the suggested provisioning plan
Done

Figure 53: The Overall Flow of the Method

performance model.

Here is a sample work flow:

1. An user submits a request to deploy a multi-tier Cloud application in an Infrastructure-as-a-Service Cloud environment. The request also describes the expected range of workloads and expected performance.

2. The application is first deployed in an over-provisioned setting.

3. While the application running in the Cloud infrastructure, its workloads and performance are monitored and the corresponding monitoring data are stored.

4. The collected workloads and performance data are used to train a cross-tier performance model.

5. The application is replicated for a set of automatic experiments which deploy the

application with different provisioning plans and measure the corresponding performance with different workloads. The goal of the automatic experiments is to learn the performance characteristics of different deployment options (e.g., virtual machine types and the number of virtual machines in a cluster).

6. The workloads and performance data collected in the automatic experiments are used to train a per-tier performance model.

7. The method explores all candidate provisioning plans and predicts the corresponding performance (for the user specified workload range) using both the cross-tier and the per-tier performance model.

8. Among all candidate provisioning plans, the one that meets the user-specified performance goal and has the lowest virtual machine instance renting cost is selected as the suggested deployment for the user.

7.2 *Prism: Performance Prediction based Cloud Application Provisioning*

We consider interactive Cloud applications such as web applications as the targeted applications in this invention. Such applications are request-driven and one request may be served by multiple components at different tiers (e.g., web servers, application servers and database servers). We use the request response time to measure the performance of applications, and use the request rate (throughput) to measure the workloads on applications.

We use the term deployment to refer to the choice of virtual machine type and cluster configuration (the number of member nodes). Our planning method consist of three techniques: 1) a prediction method that takes workloads and deployment as input, and output the predicted application performance; 2) a method that captures the changes of perceived workloads across different deployments; 3) a planning method that explores all candidate provisioning plans and outputs the optimal one.

7.2.1 The Prediction Method

The general idea of our prediction techniques is to first predict the response time for a given workload on an over-provisioned deployment (also referred to as the base deployment), and then modify the predicted response time considering changes introduced by the difference between the over-provisioned deployment and the actual targeted deployment. Correspondingly, we employ two performance models to accomplish this task, a cross-tier performance model which captures the relation between workload and response time for the base deployment, and a per-tier performance model that captures the relation between deployment changes (to the base deployment) and corresponding changes of the response time.

A cross-tier model has the following form,

$$\Theta_c(w) \rightarrow r \tag{18}$$

where w is the workload and r is the average response time of requests. The cross-tier model takes workload as input and outputs the response time on the base deployment. Note that while we use average response time to describe the techniques, our approach also supports the prediction of quantile response time (e.g., 90-th percent response time of requests). We use Kernel regression to train the cross-tier model. As a non-parametric technique, it does not specify a certain relation (e.g., linear relation) between w and r, but produces a non-linear relation between w and r that best fits the observed performance data. This flexibility is important as the actual relation between w and r may vary at different workload levels, or across different applications.

A per-tier model has the form of,

$$\Theta_p^t(w, v, c) \rightarrow r_\Delta \tag{19}$$

where t denotes the object tier, v is the virtual machine type, c is the cluster size, i.e. the number of member nodes, and r_Δ is the change of response time compared with the base deployment. The per-tier model is actually a set of models where each model is trained for a particular tier. Each per-tier model takes the workload, the type and the number of virtual machine used at the object tier as input, and outputs the changes of response time introduced by this tier over that of the base deployment. Same as the cross-tier model, we also use Kernel regression to train the per-tier model.

To predict the response time for a target deployment and a given workload, we first use the per-tier model estimate the differences of response time introduced at each tier due to the deployment differences between the target deployment and the based deployment. Specifically, the overall change of response time change R_Δ is,

$$R_\Delta \leftarrow \sum_{\forall t} \Theta_p^t(w, v(t), c(t)) \tag{20}$$

where $v(t)$ is the virtual machine type in tier t and $c(t)$ is the number of virtual machines in tier t. The final predicted response time r^* is,

$$r^* \leftarrow R_\Delta + \Theta_c(w) \tag{21}$$

where we apply the predicted response time changes to the predicted response time on the base deployment. Figure 54 illustrates the work flow of the prediction process.

The cross-tier model and the per-tier model are trained separately in two steps. The training of the cross-tier model requires only performance monitoring data on the base deployment. Note that such data can be easily collected from the base deployment when it serves user requests, which means no additional experiments are needed for data collection. Specifically, the training data set should include the request rates spanning from light workloads to peak workloads and the corresponding average response time. Off-the-shelf statistical tools can be used to train the cross-tier model (e.g. `npreg()` in R). Typically, the base deployment is over-provisioned to ensure the request response time meets the performance goal. However, our approach works on any based deployment. The base deployment is also used as contrasts to generate training data for the per-tier model.

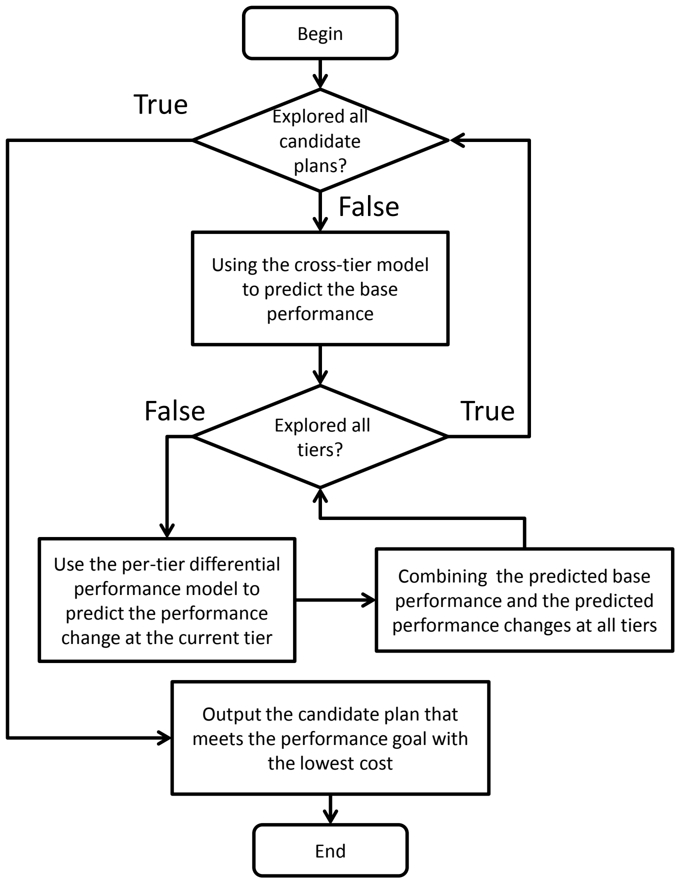

Begin
Explored all candidate plans?
True
False
Using the cross-tier model to predict the base performance
Explored all tiers?
False
True
Use the per-tier differential performance model to predict the performance change at the current tier
Combining the predicted base performance and the predicted performance changes at all tiers
Output the candidate plan that meets the performance goal with the lowest cost
End

Figure 54: The Work Flow of the Prediction Process

The per-tier models are trained in a tier-by-tier basis based on performance data collected on a series of automatic experiments (Figure 55). Specifically, we first create a duplicate of the based deployment and refer to this deployment as the background deployment. For a per-tier model on tier t, we vary the configuration of tier t on the background deployment by changing the virtual machine type and the number of virtual machines, and leave the configuration of other tiers unchanged (same as the configuration in the base deployment). This leads to mn different background deployments where m is the total number of virtual machine types and n is the maximum number of virtual machines in tier t. For each resulting background deployment (with virtual machine type $v(t)$ and virtual machine number $c(t)$ in tier t), we introduce different levels of workloads (from light level to peak level just as those in the cross-tier model training dataset) to the deployment and record the difference of response time $r_\triangle$ between the background deployment and the base deployment for each level of workload w. The workload can be generated by simple workload generation tools such as `httperf`. The resulting data points $(w, v(t), c(t), r_\triangle)$ are used to train the per-tier model Θ_p^t. Similar to the cross-tier model, Off-the-shelf statistical tools can be used to train the per-tier model.

One particularly time-consuming procedure in training the per-tier model is capturing cluster performance changes with different number of virtual machines. The virtual machine provisioning time on most Cloud platforms ranges from a few minutes to 20 minutes. As a result, adding virtual machines to a cluster one-by-one to capture the corresponding performance changes can take substantial time, especially for large clusters with many member nodes. To address this issue, we employ a multiplicative-delta learning technique that selectively performs additional experiments. Instead of adding virtual machines one-by-one, it doubles the virtual machines incremental number, if the per-tier model gives good prediction on the performance of the current cluster. If the prediction accuracy drops at certain point, it reduces the instance incremental number by half. The procedure finishes until the maximum instance number is reached. The rationale behind this technique is that

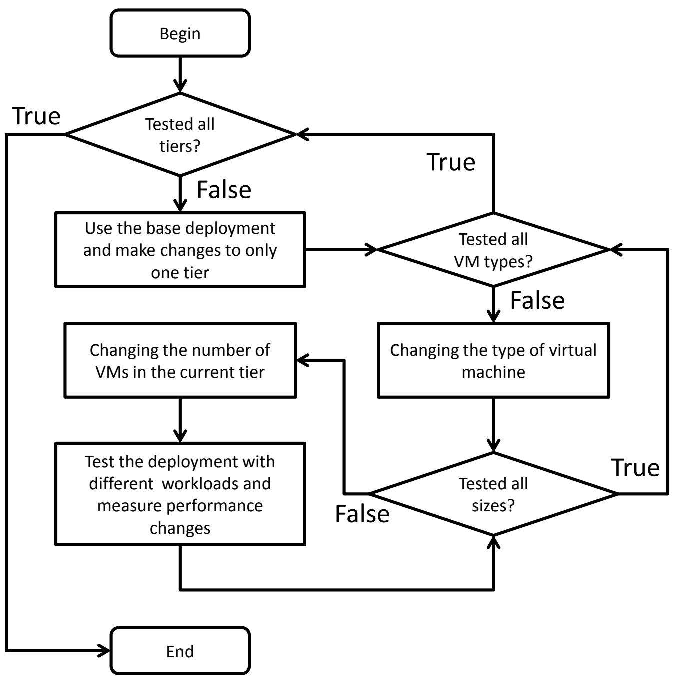

Begin
Tested all tiers?
True
False
Use the base deployment and make changes to only one tier
True
Tested all VM types?
False
Changing the number of VMs in the current tier
Changing the type of virtual machine
Test the deployment with different workloads and measure performance changes
True
False
Tested all sizes?
End

Figure 55: The Work Flow of the Automatic Experiment

most clusters implement a load-balance scheme among their member instances. As a result, the performance curve can be learned with relatively small amount of training data. Even if the cluster implements a complicated workload assignment scheme, the technique can degenerate to the original cluster performance learning procedure which intensively collects performance data points with many different size settings.

7.2.2 A Concrete Example

We now present a detailed example of deploying a web application in Smart Cloud Enterprise (SCE) to illustrate the advantage of our approach over existing techniques.

The Deployment Scenario. SCE provides 9 different types of pre-configured virtual machine instances. The configuration is defined in terms of the number of virtual CPUs, the size of virtual machine memory and the size of local storage. Different types of VMs are also associated with different hourly (renting) rate. A user wants to deploy a web application consisting of three tiers, the web server tier, the application server tier and a database tier. To deploy the web application, the user needs to decide the deployment plan for each tier which breaks down to 1) what types of VM instances to use at one tier; 2)how many VM instances to use at one tier. For the sake of this example, we assume one tier can at most utilize $N = 20$ VM instances. In addition, the user also has a performance requirement of achieving an average request response time (measured in a 10-second time window) less than 2 seconds, as long as the incoming requests rate is below a certain level, e.g., 500 requests per second. The overall deployment goal is to achieve this performance goal with minimum instance renting cost.

Experiment-based exploration. This line of methods[127] leverage the massive computing power in a datacenter to run a large number of experiments to measure the performance of an application under different deployments. While such methods may work well for applications with relative small number of candidate deployments and environments with massive (free) available computing resources, they may cause prohibitively high cost for exploring candidate deployments for multi-tier applications. In our deployment scenario, the total number of candidate deployments is $(9 \times 20)^3 = 5832000$. Even if each experiment lasts only 1 minute and all instances are charged at the lowest rate, the overall experiment would cost $\$1,204,308$ to complete.

Rules of Thumb. One may use rules of thumb to predict the unobserved performance of one deployment based on observed performance on another deployment. For instance, suppose the workloads of the web application in our scenario are CPU bounded. If one observes that a deployment consisting of only virtual machines with a single virtual CPU satisfies the performance requirement for a request rate of 250 requests per second, one may infer that a deployment consisting of only virtual machines with two virtual CPUs should satisfies the performance requirement for a request rate of 500 requests per second. Unfortunately, we find this is often not true through many experiments because the number of virtual CPUs in a virtual machine is not a good indicator for CPU performance.

Prediction techniques based on queuing model or statistical regression. These techniques are often used to train performance models for a fixed deployment and the trained performance models can predict the performance for a given workload on the fixed deployment. However, a model trained for one deployment usually cannot be used for performance prediction on another deployment. For instance, if we train a performance model based on queuing models on a deployment consisting of only virtual machines with 2 virtual CPUs, the same model is very likely to produce poor prediction results on another deployment consisting of only virtual machines with 1 virtual CPU. This is because the service time of a request usually changes across different deployments, which in turn invalidates the model on a new deployment. Regression based performance models have similar issues as the performance-workload relation changes across different deployments. Training a performance model for all candidate deployments is clearly infeasible (same as the case of experiment-based exploration).

Our approach. The central technique in our approach is building a performance model that can produce accurate performance prediction for different deployments (versus to single-deployment prediction model). First, we train a regression-based performance model on an over-provisioned deployment which we refer to as the base deployment. On SCE, such an over-provisioned deployment consists of only Platinum virtual machines (64-bit VM with 16 virtual CPUs and 16GB memory) and each tier has 20 such VMs. The training process involves feeding the base deployment with different levels of workloads and measuring the corresponding performance. The resulting performance data (average response time) and workloads are then used to train the performance model which we refer to as the cross-tier model which can predict the average response time for a certain workload on the base deployment.

Second, we train a set of models that captures the performance changes introduced by using different VM types and different number of VMs at each tier. This process is performed on a tier-by-tier basis with a outer loop and an inner loop. The outer loop deals with one tier at a time and the inner loop captures the performance changes brought by deployment changes at one tier. The outer loop first pick the web server tier for manipulation.

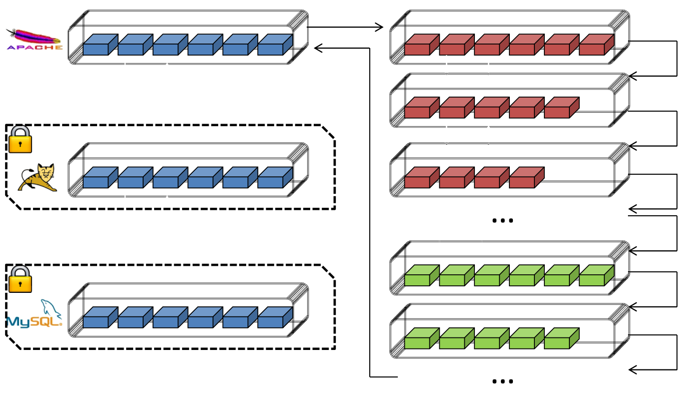


Figure 56: Training Per-Tier Models

Within the corresponding inner loop, we first change the types of VMs from Platinum to 64-bit Gold (8 virtual CPUs and 16GB memory) at the web server tier, and measure the difference between performance on the new deployment and that on the base deployment given different levels of workloads. We then reduce the number of VMs at the web server tier one-by-one, and measure the difference between performance on the resulting deployment and the base deployment. Note that while we change the VM type and number at the web server tier, the other two tiers, the application server tier and the database tier, are left unchanged (same as those in the base deployment).

Similarly, we change the VM type to 64-bit Silver (4 virtual CPUs and 8GB memory) and vary the number of VMs at the web server tier. For each resulting deployment, we measure the difference between performance on the new deployment and that on the base deployment given different levels of workloads. We repeat this process until we have tried all VM types on the web server tier. The collected performance difference data allow us to train a web server tier model (Equation 7.2.1) that predicts the performance changes introduced by deployment changes (i.e., VM type and number) at the web server tier of the base deployment. Up to now, the first round of the outer loop finishes. Figure 56 illustrates the training process.

For the second round, we change the deployment of the application server tier. Note that this time we keep the web server tier and the database tier the same as those in the base deployment. We follow the same procedure to change the VM type and the VM number and measure the corresponding performance difference. The generated data leads to a application server tier model that predicts the performance changes introduced by deployment changes at the application server tier of the base deployment. Similarly, the final round works on the database tier and produces a database tier model that predicts the performance changes introduced by deployment changes at the application server tier of the base deployment. The three models trained are referred to as per-tier models.

We now can predict the performance of an arbitrary deployment based on the cross-tier model and per-tier models. Suppose we want to know the average response time on a deployment consisting of 5 Bronze VMs (2 virtual CPUs and 4GB memory) at the web server tier, 10 Silver VMs (4 virtual CPUs and 8GB memory) at the application server tier and 20 Gold VMs (8 virtual CPUs and 16GB memory) at the database tier when given a workload of 500 requests per second. We first use the cross-tier model to predict the average response time for the given workload (500 request/second). Note that the predicted response time which we refer to as the base response time is for the base deployment. Next, we apply the web server tier model (a per-tier model) to predict the changes of response time contributed by the deployment changes at the web server tier (compared with that of the base deployment). As 5 Bronze VMs have much less processing power compared with 20 Platinum VMs in the base deployment. The predicted response time change is very likely to be a positive value. Similarly, we also apply the application server tier model and the database tier model to obtain the predicted response time changes at the corresponding tiers. Finally, we sum up the base response time and the three predicted response time changes at different tiers together to obtain the predicted response time for the given deployment. Figure 57 illustrates the prediction process. Note that this example does not show the detail of handling cross-Deployment workload changes described in Section 7.2.3.

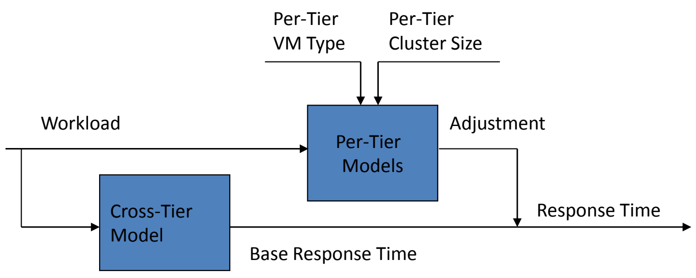


Figure 57: Illustration of The Prediction Process

7.2.3 Capturing Cross-Deployment Workload Changes

The above prediction method makes an implicit assumption that the actual workloads perceived at each tier do not change across different deployments. This assumption, however, may not hold for many Cloud applications. The perceived workload at a tier may not be the same as the workload introduced to the application due to prioritization, rate limiting mechanisms implemented at different tiers. For instance, an application may drop certain low-priority requests when a certain tier becomes performance bottleneck, which in turn causes the change of workload at other tiers. Even for applications without prioritization mechanisms, a bottleneck tier may limit the overall system throughput and introduce changes to the workload on other tiers.

Performance prediction without considering such workload changes may lead to significant prediction accuracy loss. As another example, a database tier of a web application configured with a single low-end virtual machine can be a performance bottleneck when the web application is fed with a peak workload w_p. As a result, the actual workloads perceived at each tier w' is often less than w_p as a certain amount of requests are queued due to database overloading. Clearly, using the data (w_p, v, c, r_Δ) for training would introduce error to the per-tier model. To address this issue, we introduce a throughput model Θ_h^t for a tier t with the following form,

$$\Theta_h^t(w, v, c) \rightarrow w' \tag{22}$$

where w' is the actual workload perceived by all tiers. When making performance predictions, we apply the throughput model to obtain the predicted workload at each tier, and use the lowest predicted workload as the input of the per-tier model. Specifically, with the throughput model, the per-tier model has the following form,

$$\Theta_p^t(\min_{\forall t} \Theta_h^t(w, v(t), c(t)), v, c) \rightarrow r_\Delta \tag{23}$$

where we replace the input workload w with the actual workload predicted by the throughput model. We also use Kernel regression to train the throughput model. Note that the data used for training the throughput model is (w, v, c, w') and w' can be easily measured by counting the number of responses within a time window.

7.3 Supporting Request-Mix Awareness

Application workloads often consist of requests of different types and requests of different types often introduce different processing overheads. For instance, for eBay-like applications, bidding requests usually incur higher costs than browsing requests do as bidding often involves database transactions. As a result, even if two workloads have the same request rate, they may result in very different resource consumption and performance if the composition of requests are very different, e.g., a 100 request/second workload with 20% bidding requests and 80% browsing requests versus another 100 request/second workload with 80% bidding requests and 20% browsing requests.

Performance oriented provisioning planning for application with heterogeneous per-request costs requires fine-grain definition of workloads with information on the composition of requests. Accordingly, performance prediction should also consider the composition of requests, a feature we refer to as request-mix awareness. We next describe the details of supporting request-mix awareness in our performance models.

To support request-mix-aware prediction, we first introduce a set of new inputs which describe the request composition of a workload. Specifically, we replace the workload w (scalar) with a vector $R = r_1, r_2, \ldots, r_k$ where r_i is the rate of requests of type i. For the brevity of discussion, we still predict the overall response time for all requests. Note that our techniques can be directly used to predict the response time for a specific type, or a set of types, of requests by simply using the corresponding response time (of the specific type, or a set of type, of requests) to train models.

Recall that training a model that is oblivious to request composition requires only generating workloads with different request rates, i.e., the model input (request rate) is a scalar. Training a request-mix-aware model, however, would require much more performance measurement (training) data with different compositions of types of requests due to the extra degrees of freedom introduced by per-request-type workloads, i.e. the model input (per-type request rate) is a vector. As a result, this would significantly increase the experiment time and make the model training process very expensive and infeasible. For example, suppose we have 20 different types of requests and we measure request rates in 10 different levels (e.g., 0-100, 100-200, 200-300, etc.), the ideal training data would include all compositions of per-type request rates (10^{20} different workloads) which is clearly impractical. Note that even though we often do not need the ideal set of data to achieve reasonable prediction accuracy, e.g., a 10% subset of the ideal training data (randomly selected) may be sufficient, a small percentage of such a large dataset (e.g., 10% of 10^{20}) is still practicaly infeasiable to generate.

7.3.1 Efficient Training of Request-Mix-Aware Models

We next describe a technique that can substantially reduce the needed experiment time. The basic idea is to automatically find cost relationship between different requests, e.g., request A and B have similar cost, or the cost of request A is about 2 times higher than that of request B. Such cost relationships allow us to map the original workload vector into a new workload vector with much smaller number of dimensions, which in turn greatly reduces the amount of training data needed to reflex different workload compositions. For the previous example, if we can group 20 different types of requests into 2 general types (e.g., transactional and non-transactional), we effectively reduce the number of compositions in the ideal training dataset from 10^{20} to 10^2.

To illustrate the advantage of our technique, we first introduce two alternative techniques. One technique is to remove requests with trivial overheads from the performance model. For instance, HTTP requests such as `home` in RUBiS asking for a small static html file (often cached) from the web server. However, this technique cannot substantially reduce the dimension of the model input vector as such low-cost requests often contribute to a very limited portion of the overall workloads (e.g., $< 1\%$ in RUBiS). A slightly advanced technique is to cluster requests into different groups where requests within the same group have similar overheads. It reduces the diemsnion of the model input from the number of request types to the number of clusters. Despites its intuitiveness, it has a serious drawback due to the binary true-false relation it poses to pairs of request types. Consider a pair of request types A and B. Requests of type A and B both cause the database server to perform the same `SELECT` operation and the only difference is that the `SELECT` operation is executed once for A but twice for B, i.e. a request of type B is apprxomately two times more expensive than a request of type A. If A and B are clustered into different groups with fine clustering granularities, the total number of groups can be quite large as only requests with very similar overhead are grouped together. However, if A and B are clustered into the same group, different compositions of type A and B requests may lead to very different workloads due to overhead difference between A and B, even if the total number of requests of this general type may be the same.

Our technique flexibly captures the cost relation between different request types. Specifically, for requests of the same group, we capture their relative overhead with a linear system. For the previous example, the total workload introduced by requests of type A and B $W_{A,B} = N_A + 2N_B$ where $N(\cdot)$ is the request number of a certain type. Formally, we linearly project the original workload vector $\vec{W}$ defined in a high dimensional space into a new workload vector $\vec{W}*$ defined in a lower dimensional space.

The main difficulty in this projection process is to ensure that the new $\vec{W}*$ can accurately represent the true workload so that the our performance model can provide good prediction. Achieving this goal, however, involves two challenges. First, how to evaluate the quality of a projection π? Although it is possible to apply π to get $\vec{W}*$ from $\vec{W}$, and compare the prediction accuracy of the performance model trained with $\vec{W}*$ and that of the model trained with $\vec{W}$, such an approach is also prohibitively expensive given the computation cost of model training. Second, how to efficiently explore and evaluate different projections to find an optimal one? Brute force approaches that explore all possible projections are clearly infeasible due to the countless number of possible projections.

7.3.1.1 Efficient Evaluation of A Projection

To address the first challenge, we must find an approach that can evaluate the quality of a projection without actually training a performance model based on the projected model input. We choose to use mutual information between the projected model input and the corresponding response time as the metric for evaluation, i.e., $I(R, \vec{W}*)$ where R is the response time and $\vec{W}*$ is the projected model input. Mutual information[121] of two random variables is a quality that measures the mutual dependence of the two random variables. Formally, the mutual information of two discrete random variables X and Y can be defined as,

$$I(X,Y) = \sum_{y \in Y} \sum_{x \in X} p(x,y) \log(\frac{p(x,y)}{p(x)p(y)}) \tag{24}$$

Mutual information measures the information that X and Y share: it measures how much knowing one of these variables reduces uncertainty about the other. For example, if X and Y are independent, then knowing X does not give any information about Y and vice versa, so their mutual information is zero. At the other extreme, if X and Y are identical then all information conveyed by X is shared with Y: knowing X determines the value of Y and vice versa.

Fano's inequality suggests that we can find the optimal projection π by maximizing

$I(R, \vec{W}*)$. This result determines a lower bound to the probability of error when estimating a discrete random variable R from another random variable $\vec{W}*$ as

$$Pr(r \neq \hat{r}) \geq \frac{H(R|\vec{W}*) - 1}{log(|R|)} = \frac{H(R) - I(R, \vec{W}*) - 1}{log(|R|)} \tag{25}$$

Hence, when the mutual information between R and $\vec{W}*$ is maximized, the lower bound on error probability is minimized. Therefore, mutual information serves as a good indicator for the quality of projection, because the higher the mutual information is, the higher predictability of the model built based on the projected model input is.

7.3.1.2 Efficient Search for An Ideal Projection

Since we use $I(R, \vec{W}*)$ to measure the quality of a projection and the ideal projection is the one that maximizes $I(R, \vec{W}*)$, the search for an ideal projection can be formulated as optimization problem defined as follows,

$$\pi = \arg\max_{\pi} I(R, \vec{W*}(\pi)) \tag{26}$$

where $\vec{W*}(\pi)$ is the resulting model input generated by using projection π. As a result, we can perform gradient ascent on I to find the optimal projection as follows,

$$\pi_{t+1} = \pi_t + \eta \frac{\partial I}{\partial \pi} = \pi_t + \eta \sum_{i=1}^{N} \frac{\partial I}{\partial w_i} \frac{\partial w_i}{\partial \pi} \tag{27}$$

$I(R, \vec{W}*)$ can be written as,

$$I(R, \vec{W}*) = \sum_{r \in R} \int_{w*} p(r, w*) log \frac{p(r, w*)}{p(r)p(w*)} dw* \tag{28}$$

We use the data collected on the base deployment to perform the search for the optimal projection. Since we use workload and performance data collected from the base deployment during the actual application runtime, there is no additional cost in generating training data for the searching of the optimal projection. In addition, as the cost relationship between different types of requests is independent of deployments, we apply the learned π to the training process of the reference model.

To determine the number of dimensions in the projected workload vector $\vec{W}*$, we let user choose the acceptable time length of automatic experiments and then use this information to derive the dimensions of $\vec{W}*$. For instance, suppose a user specifies that the experiment of each deployment should not exceed 30 minutes. If the performance measurement of a given workload can be done in 30 seconds, the total number of workload compositions we can test on one deployment is 60 ($60 \times 1/2 = 30$). If a 10% random sampling of workload composition is good enough for model training and there are 5 diferent levels for the request rate, we then have a total population of 600 ($60/0.1 = 600$) workload compositions which approximately correspones to a dimenion of 4 in $\vec{W}*$ ($5^4 = 625 \approx 600$). Note that we can also let user specify a high level cost requirement for model building, e.g., the maximum time for experiment or even the total monetary cost for experiment, and we then derive the dimension of $\vec{W}*$ based on the above process, the number of deployments needed to test for collecting data and the virtual instance pricing policy.

7.3.2 Provisioning Planning

With the prediction model described above, finding the optimal provisioning plan for an application is straightforward. It requires only exploring all candidate provisioning plans and estimating the cost (monetary cost such as virtual machine renting fee which can be easily computed based on the pricing policy of a Cloud platform) and performance (obtained by our prediction method) of each candidate plan. The optimal plan is the one with the lowest cost and performance that satisfies the performance goal. As the cost estimation and performance prediction introduces trivial computational cost, the overall search process can often be completed within a few seconds. In addition, the performance prediction model, once trained, can be repeated used for different planning tasks with different performance goals.

Table 2: Virtual Machines in SCE

Name in SCE	VM Type	vCPU	Memory(GB)	Disk(GB)	OS	Hourly Rate
Copper32	A	1	2	60	RHEL32bit	\$0.19
Copper64	B	2	4	60	RHEL64bit	\$0.4
Gold32	C	4	4	350	RHEL64bit	\$0.5
Silver64	D	4	8	1024	RHEL64bit	\$0.61
Gold64	E	8	16	1024	RHEL64bit	\$0.94
Platinum64	F	16	16	2048	RHEL64bit	\$ 1.84

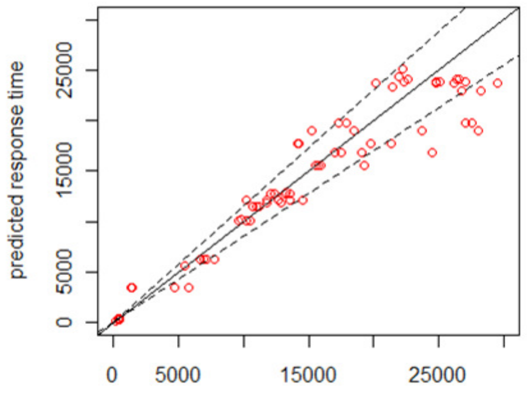


Figure 58: Performance Prediction Accuracy

7.4 Evaluation

We conduct our experiment in IBM's Smart Cloud Enterprise (SCE), a production Infrastructure-as-a-service platform similar to Amazon's EC2. SCE offers a variety of virtual machines with different capacities and hourly rates as shown in Table 2. We use a 3-tier RUBiS [8] as our application benchmark.

Figure 58 shows the prediction accuracy of our approach where the x-axis is the observed average response time and the y-axis is the predicted average response time. For each data point, the x value is the observed average response time for a certain workload and the y value is the predicted average response time for the same workload. We use Absolute Percentage Error (APE) to measure the accuracy of prediction where $APE = \frac{|ObservedResponseTime - PredictedResponseTime|}{ObservedResponseTime}$. The dashed lines indicate the 0.1 APE range. Most data points in the figure are close to the line $y = x$, which indicates that the predicted response time is close to the observed value. Overall, the average APE of all data points is

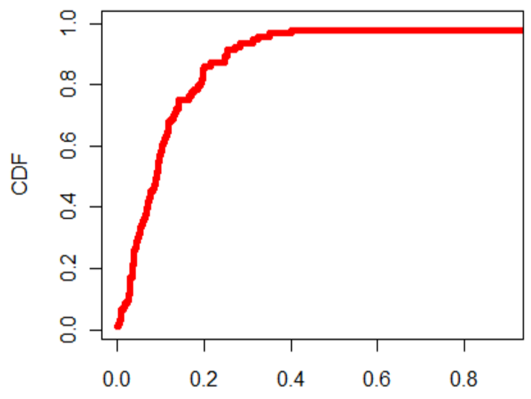


Figure 59: CDF of Performance Prediction Error

0.15371. Figure 59 shows the distribution of APE of all data points in Figure 58. About 80% of data points have APE less than 0.15.

We compare our approach with an utilization based approach in Figure 60 where *differential* denotes our approach and *utilization* represents the utilization based approach. The utilization based prediction approach utilizes queuing theory and adopts a simple assumption that the relationship between resource utilization and the response time is constant across different deployments. This approach first trains workload-utilization models for different types of virtual machines, and then trains a utilization-response time model on one type of virtual machines. To predict response time for a given workload w, it first uses the workload-utilization model to get the corresponding utilization which is then used to predict the corresponding response time with the utilization-response time model. Clearly, the utilization based approach achieves much worse prediction accuracy compared with our differential prediction approach. About 60% of its predictions has an APE larger than 0.2.

To illustrate why the utilization based approach[103] does not perform well, we use Figure 61 to show the relationship between response time and the corresponding CPU utilization across different types of virtual machines runnning MySQL. The x-axis shows the increasing utilization and the y-axis shows the average response time. The curves from right to left represents virtual machines of type A, B, C, E, F (each type of VM has two times more vCPUs than the previous one). While the CPU utilization of type A VM spans

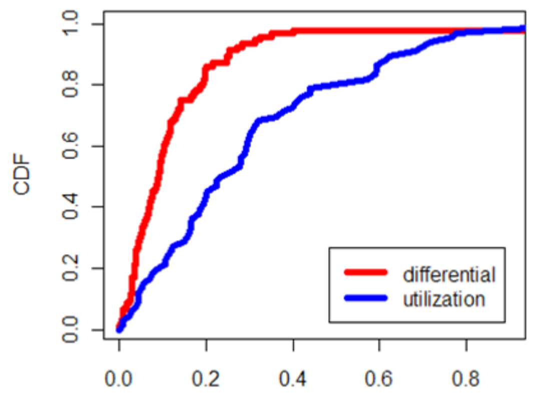


Figure 60: Performance Prediction Accuracy

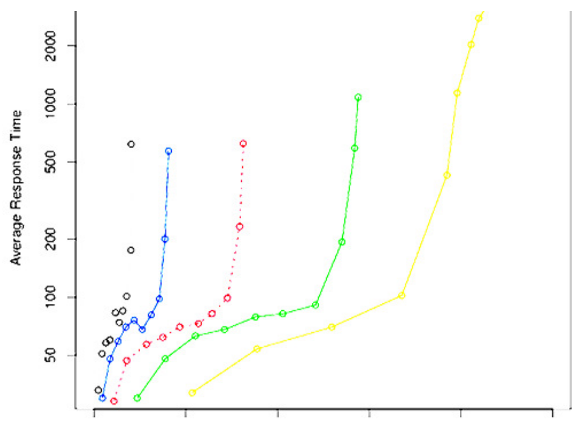


Figure 61: Performance Prediction Accuracy

from 20% to almost 100%, the CPU utilization of other types of VMs spans over smaller and smaller ranges with increasing number of virtual CPUs. We find that the reason for this reducing range of CPU utilization is the skewed distribution of CPU utilizatoin acorss vCPUs. For VMs with more than 1 vCPU, usually only the first vCPU is highly utilized while the rest of vCPUs are under utilized. As a result, the assumption of constant utilization-resonse time does not hold, which in turn leads to poor prediction performance of the utilization based approach.

We also performed case study to evaluate the overall performance of our Cloud application provisioning approach. We use a SLA defined as follows: For a maximum workload of 2400 user sessions, 75% requests should have a response time less than 600ms. Our approach produces a provisioning plan A2A2B4 (2 type A VMs running the web server, 2 type A VMs running the application server, 4 type B VMs running the database server) which leads to an SLA violation ratio of 0.04% and an overall cost of $2.36 per hour. With the utilization based approach, the resulting provisioning plan is E1E1E1 (1 type E VM running the web server, 1 type E VM running the application server, 1 type E VM running the database server) which leads an SLA violation ratio of 2.34% and an overall cost of $2.82 per hour. Finally, we also evaluate the performance of an instance driven provisioning. It first uses a A1A1A1 deployment to test the maximum workloads such a deployment can run without violating the SLA. It then simply increasing the number of instances at each tier to reach the throughput specified in the SLA assuming that the throughput of the resulting deployment increases linearly with the number of instances. The instance-drive approach produces a provisioning plan of A8A8A8 which leads to a SLA violation ratio of 0.03% and a high overall cost of $4.56 per hour, a cost almost doubled compared with our approach.

7.5 Related Work

Existing techniques used to plan multi-tier web application provisioning can be categorized into three classes.

Experiment based exploration. This line of works[127] use automatic configured experiments to test the application performance of different provisioning plans. As these works often need to explore the entire configuration space to produce definite results, the corresponding cost is fairly high in terms of both resource consumption and execution time.

Rules of thumb. These techniques use intuitive observation and heuristics to predict performance and plan provisioning. For instance, if a VM with a single vCPU can achieve throughput X, then a VM with 2 vCPUs should produce a throughput of 2X. These techniques often overlook complex performance behaviors of applications, virtual machines and clusters, which often lead to poor prediction accuracy. Due to its simplicity, rules of thumb also do not cover certain provisioning options. For instance, it is not clear how to develop a rule of thumb for predicting the growth of request response time given increasing throughput for an arbitrary application.

Single-deployment performance modeling. Performance modeling is often used to capture the relation between workloads and performance for a specific deployment. Existing approaches along this direction often apply queuing models[108] and regression techniques[103]. Queuing model based approaches often use instrumentation at the middleware level or the operating system level to obtain critical model parameters such as per-request service time. This often limits the applicability of these approaches in Cloud environment where middleware or OS instrumentation may not be a valid option for Cloud users. Other works employ assumptions or approximations to apply analysis techniques such as mean value analysis (MVA),which may limit the accuracy of the model and its prediction performance. Regression based approaches utilize statistical regression techniques to build prediction models which captures the relation between workload and performance. Compared with queuing model based approaches, they do not explicitly model the internal processing and waiting mechanisms of an application, but simply capture the fundamental relationship between workload and performance. As a result, they do not require instrumentation to build models, and introduce little assumption or approximation.

These approaches build models for a specific hardware/software deployment, e.g., fixed machines and cluster configurations, and focus on the impact of workload changes on performance. The resulting models often produce poor prediction results on a different hardware/software deployment. On the contrary, our approach not only considers workload changes, but also deployment changes, e.g. what if using machine A instead of machine B to run the database server. This cross-deployment feature is important for Cloud application provisioning due to the large number of available deployment options (e.g., different virtual machines types and different cluster configurations). Another distinct feature of our approach is that it utilize a per-tier model to capture the performance difference introduced by deployment changes at each tier. This allows us to predict performance changes for any combination of deployment changes at different tiers without collecting performance data from the corresponding deployment, which saves tremendous amount of experiment time and cost.

CHAPTER VIII

CONCLUSIONS AND FUTURE WORK

This dissertation makes three unique contributions. First, we study the problem of accurate and efficient local-to-global state aggregation in distributed state monitoring. At the global violation detection level, we develop window based state monitoring (Chapter 4) to prevent monitoring data noises to trigger unnecessary state violations. Furthermore, we devise a distributed state monitoring algorithm that utilizes distributed monitoring windows to achieve significant monitoring communication reduction which in turn saves considerable CPU resources on communication endpoints such as monitors and coordinators. At the local state collection level, we develop violation likelihood based sampling technique (Chapter 5) that dynamically tunes sampling intensities based on the likelihood of detecting important results, which allows a flexible tradeoff between sampling cost and monitoring accuracy.

Second, rather than assuming perfectly reliable monitoring environments in Cloud datacenters, we consider various dynamics such as communication issues and node failures that are common in a virtualized datacenters. These dynamics not only widely exist across virtualized datacenters due to performance interferences, application scale and management complexities, but also potentially introduce considerable monitoring errors to state monitoring approaches that depend on reliable communication and always-online monitoring nodes. We propose a robust state monitoring algorithm (Chapter 6) that not only continuously annotates monitoring results with accuracy estimation, but also adapts to long-term communication issues or node offline events. Overall, it maximizes the utility of monitoring data even when they are incomplete or error-prone and helps Cloud application performance management tasks such as auto-scaling to achieve better results.

Third, we also propose a set of techniques to address several important issues on the distributed coordination model, including optimizing monitoring communication at multi-task level, elasticity of the monitoring system and utilizing state monitoring data to simplify application provisioning. We present REMO (Chapter 2) to provide multi-tenancy support for different state monitoring service users through safeguarding per-node level monitoring resource consumption and exploring cost sharing opportunities among different monitoring tasks. We develop Tide (Chapter 3) to provide the elasticity that state monitoring servers require to keep up with the on-demand, highly dynamic Cloud workloads. We also introduce Prism (Chapter 7) to show that state monitoring data can also be used to support advanced Cloud management tasks.

When looking at different ways to realize state monitoring, there are three general classes of approaches.

- *Centralized comprehensive data collection, processing and analysis* [20, 31, 32]. This type of approaches collect all monitoring related data to a central repository where data processing and analysis are performed. Note that such centralized approaches still need to collect monitoring data from distributed nodes.

- *Centralized selective data collection, processing and analysis* [56, 54, 98, 29]. The second type of approaches selectively collect monitoring data from a distributed system or application based on the goal of monitoring and analysis. They still perform centralized processing and analysis on the collected data.

- *Distributed data collection, processing and analysis* [39, 97, 60, 11, 57]. The final class of approaches adopt a fully distributed monitoring paradigm where monitoring data collection, processing and analysis are all distributed across monitoring nodes. Techniques along this line often require distributed algorithms designed for different monitoring and analysis tasks.

Techniques introduced in this dissertation primarily focus on the second and third monitoring paradigms. Specifically, window based state monitoring (Chapter 4) and robust state monitoring (Chapter 6) contribute to the algorithm design for distributed monitoring and analysis systems. Violation likelihood based state monitoring (Chapter 5) is useful for both centralized selective monitoring systems and distributed monitoring systems. Resource-aware state monitoring (Chapter 2) and self-scaling state monitoring (Chapter 3) can be applied to all three types of monitoring approaches as they both focus on multi-tenancy of monitoring data collection/processing. However, they may provide additional performance benefits in the second and third types of approaches due to their distributed nature.

8.1 Ongoing Research, Related Work and Open Problems

There are several interesting ongoing research directions that we intend to pursue. First, our approaches focus on simple state monitoring form where monitoring data are evaluated with simple thresholds and time windows. Furthermore, we use simple aggregation operators such as sum and average for aggregating distributed data collected from different monitors. It would be interesting to explore approaches that support advanced evaluation of state violations (e.g., statistical hypothesis testing) and other aggregations (e.g., statistical sketch). Second, state monitoring techniques introduced in this paper are often tailored towards a specific monitoring requirement. For instance, our window based state monitoring works best for users who are interested in continuous violation events (versus non-continuous ones such as percentage of violation events within a time window). As another example, our violation likelihood based state monitoring is most useful for monitoring tasks with certain level of tolerance to miss-detection. Hence, it is worthwhile to explore new monitoring approaches (e.g., meta monitoring engine) that can generalize different monitoring requirements and can be self-tuned to meet diverse monitoring needs. Finally, although it is possible to employ all our approaches simultaneously at the levels of local state collection and distributed violation detection, we have not yet studied the integration of all our techniques at different levels. It would also be an interesting topic to study the value-add opportunities in such integration as well as cases with possible diminishing marginal benefits.

While there are few works dedicated on state monitoring, there are a number of previous monitoring works related with the problem we study in three research areas: *Sensor Network*, *Distributed Aggregation* and *Resource and Performance Management Oriented Monitoring*.

Sensor Network. A number of existing works in sensor networks use correlation to minimize energy consumption on sensor nodes[98, 29]. Our work differs from these works in several aspects. First, these works often leverage the broadcast feature of sensor networks, while our system architecture is very different from sensor networks and does not have broadcast features. Second, we aim at reducing sampling cost while these works focus on reducing communication cost to preserve energy. Third, while sensor networks usually run a single or a few tasks, we have to consider multi-task correlation in large-scale distributed environments. Finally, some works (e.g., [29]) make assumptions on value distributions, while our approach makes no such assumptions.

Distributed Aggregation. Distributed data aggregation [105] has been an active research area in recent years. Researchers have proposed algorithms for efficiently performing continuous monitoring of top-k items [19], sums and counts [86] and quantiles [34], skylines [35], joins [124] and max/min values [99]. Problems addressed by these work are quite different from ours. While these work study supporting different operators, e.g. top-k and sums, over distributed data streams with guaranteed error bounds, we aims at detecting whether a simple aggregate (e.g., sum and average) of distributed monitored values violates constraints defined in value and time.

cSAMP [96] is network traffic flow monitoring system that minimizes monitoring cost with flow-based sampling, has-based coordination and network-wide optimization. Compared with this work, our approach aims at performing efficient distributed state monitoring over performance metrics of systems or applications.

Jain and et al.[55] studies the impact of hierarchical aggregation, arithmetic filtering and temporary batching in an unreliable network. They propose to measure the degree of inaccuracy based on the number of unreachable monitoring nodes and the number of duplicated monitoring messages caused by DHT overlay maintenance. While this work provides insight for understanding the interplay between monitoring efficiency and accuracy given message losses, it also has several limitations such as not considering delay and difficulties in measuring monitoring accuracy. Our work is complementary to [55] as we try to move forward the understanding of monitoring reliability by studying accuracy estimation and self-adaptation in state monitoring.

Resource and Performance Management Oriented Monitoring. Wang and et. al. [119, 117] studied a series of entropy-based statistical techniques for reducing the false positive rate of detecting performance anomalies based on performance metric data such as CPU utilization with potential data noises. Compared with this work, our approach uses relatively simple noise filtering techniques rather than sophisticated statistical techniques to reduce monitoring false positives, but focuses on devising distributed window based state monitoring algorithms that minimize monitoring related communication, which in turn reduces monitoring related CPU consumption. Note that our approach may still be used to efficiently collect monitoring data which are fed to statistical techniques for sophisticated application performance monitoring data analysis. For instance, one may use our approach avoid collecting data with trivial statistical significance by using a low global threshold and a short time window to filter such data and save monitoring data collection overheads.

Viswanathan and et. al. [109] recently proposed ranking windows of monitored metrics based on their probability of occurrence computed based on the false positive/negative rates for monitoring application performance anomalies. While this approach provides useful prioritizing of monitoring alerts generated by a collection of performance monitoring tasks, especially when administrators are overwhelmed by a large number of monitoring alerts, our approach aims at improving the monitoring accuracy of each individual monitoring task with the existence of data noises or communication issues.

Wang, Kutare and et. al. [118, 66] proposed a flexible architecture that enables the tradeoff between monitoring/analysis costs and the benefits of monitoring/analysis results for web application performance analysis and virtual machine clustering. The architecture utilizes reconfigurable software overlays (Distributed Computation Graphs (DCGs)) which undertakes monitoring data collection, exchange and processing. While this work considers monitoring cost in terms of capital cost of dedicated monitoring hardware or software, our approach considers primarily CPU resource consumption related to monitoring communication or data collection. Furthermore DCGs focus on designing a flexible monitoring/analysis architecture. In contrast, we aim at developing concrete distributed monitoring algorithms that minimizes monitoring communication or data collection for a specific form of monitoring (state monitoring).

This dissertation research presents only one step towards a truly scalable and customizable MaaS solution. Many issues need to be investigated in depth for MaaS to be a successful service computing metaphor for Cloud state management. Here we list a few examples of these important open problems in providing MaaS.

Monitoring Heterogeneity. While this dissertation research focuses primarily on the most widely used monitoring form, state monitoring, which tracks the monitored state change based on numerical metric values, there are other commonly used monitoring forms. For instance, log monitoring is also important for tracking anomalies and locating root causes, especially in distributed environments. As another example, flow-based monitoring is useful for distributed multi-tier applications. Flow-based request processing monitoring tracks the execution of a request across nodes and can be used to quickly isolate performance problems or bugs. How to support these forms of monitoring in an efficient and scalable manner? What are the implications of supporting these monitoring forms to the monitoring infrastructure? These are all open problems waiting to be solved.

Support Smart Cloud Management. Automation is the key for Cloud management given the complexity of the Cloud infrastructure, platforms and applications. While we have investigated the possibility of utilizing performance monitoring data to automate performance-driven Cloud application provisioning, this is only a first step towards smart Cloud management. MaaS should explore other automation opportunities based on the rich set of monitoring data it collects. For instance, Cloud applications evolve over its lifetime in the form of reconfiguration, feature enrichment, bug fixing, new functionalities implementation, etc. Many often experience fast evolution pace due to quick release cycles (e.g., mobile application backends). Managing Cloud application evolution involves many challenges such as configuration management. Since many Cloud applications often use common software components (e.g., application servers), and even share certain tiers (e.g., database), it is also possible to develop intelligent techniques that detect misconfigurations through analysis of relevant configuration data across applications. Other challenging problems along this line also include disturbance-free patching scheduling, performance prediction for software/platform changes, automatic bug localization, etc.

Security and Privacy. Cloud promotes efficient and flexible computing paradigms through resource sharing and workload consolidation, which also brings new challenges in privacy and security. For instance, VMs running on the same host are vulnerable to performance attacks that exploit the limitation of current hypervisors in performance isolation. VMs may also expose sensitive information to infrastructure service providers as the latter has complete access to their VMs. New techniques such as cross-VM memory sharing further improve cost-effectiveness of Cloud services, but may open doors for new privacy and security threats that exploit page sharing mechanisms to hurt system performance or even cause privacy breach when combined with other attack techniques. With increasing VM activities per host, VM related traffics within a server box essentially become unmonitored dark regions where new forms of attack may rise. MaaS should also provide new monitoring functionalities to address such security and privacy issues in Cloud environments. In addition, MaaS itself should incorporate new designs with build-in monitoring data security and privacy support.

REFERENCES

[1] Amazon Elastic Compute Cloud, CloudWatch, http://aws.amazon.com/cloudwatch/.

[2] The Internet2 Observatory Data Collections, http://www.internet2.edu/observatory/archive.

[3] WorldCup Trace, http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[4] "Auto scaling." http://aws.amazon.com/autoscaling/.

[5] "Enomaly homepage." http://www.enomaly.com/.

[6] "Microsoft System Center." http://www.microsoft.com/systemcenter.

[7] "oVirt home page.." http://ovirt.org/.

[8] "Rubis." http://rubis.ow2.org/.

[9] "Visual evidence of amazon ec2 network issues." https://www.cloudkick.com/blog/2010/jan/12/visual-ec2-latency/, 2010.

[10] ABADI, D. J., MADDEN, S., and LINDNER, W., "Reed: Robust, efficient filtering and event detection in sensor networks," in *VLDB*, 2005.

[11] AGRAWAL, S., DEB, S., NAIDU, K. V. M., and RASTOGI, R., "Efficient detection of distributed constraint violations," in *ICDE*, 2007.

[12] AMAZON, "Amazon web service(aws)." http://aws.amazon.com.

[13] AMAZON, "Auto scaling." http://aws.amazon.com/autoscaling/.

[14] AMAZON, "Amazon elastic computer cloud(amazon ec2)," 2008.

[15] AMINI, L., JAIN, N., SEHGAL, A., SILBER, J., and VERSCHEURE, O., "Adaptive control of extreme-scale stream processing systems," in *ICDCS*, 2006.

[16] ANDERSEN, D. G., FRANKLIN, J., KAMINSKY, M., PHANISHAYEE, A., TAN, L., and VASUDEVAN, V., "Fawn: a fast array of wimpy nodes," in *SOSP*, pp. 1–14, 2009.

[17] APACHE, "Hbase." http://hbase.apache.org/.

[18] ARLITT, M. and JIN, T., "1998 world cup web site access logs." http://www.acm.org/sigcomm/ITA/, August 1998.

[19] BABCOCK, B. and OLSTON, C., "Distributed topk monitoring," in *SIGMOD*, 2003.

[20] BAHL, P., CHANDRA, R., GREENBERG, A. G., KANDULA, S., MALTZ, D. A., and ZHANG, M., "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *SIGCOMM*, pp. 13–24, 2007.

[21] BHATIA, S., KUMAR, A., FIUCZYNSKI, M. E., and PETERSON, L. L., "Lightweight, high-resolution monitoring for troubleshooting production systems," in *OSDI*, pp. 103–116, 2008.

[22] BORKOWSKI, J., "Hierarchical detection of strongly consistent global states," in *ISPDC/HeteroPar*, pp. 256–261, 2004.

[23] BORKOWSKI, J., KOPANSKI, D., and TUDRUJ, M., "Parallel irregular computations control based on global predicate monitoring," in *PARELEC*, 2006.

[24] BULUT, A. and SINGH, A. K., "A unified framework for monitoring data streams in real time," in *ICDE*, 2005.

[25] BURROWS, M., "The chubby lock service for loosely-coupled distributed systems," in *OSDI*, pp. 335–350, 2006.

[26] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., and GRUBER, R., "Bigtable: A distributed storage system for structured data," in *OSDI*, pp. 205–218, 2006.

[27] CHASE, J. S., ANDERSON, D. C., THAKAR, P. N., VAHDAT, A., and DOYLE, R. P., "Managing energy and server resources in hosting centres," in *SOSP*, pp. 103–116, 2001.

[28] CHASE, J. S., IRWIN, D. E., GRIT, L. E., MOORE, J. D., and SPRENKLE, S., "Dynamic virtual clusters in a grid site manager," in *HPDC*, pp. 90–103, 2003.

[29] CHU, D., DESHPANDE, A., HELLERSTEIN, J. M., and HONG, W., "Approximate data collection in sensor networks using probabilistic models," in *ICDE*, p. 48, 2006.

[30] CLARK, C., FRASER, K., HAND, S., HANSEN, J. G., JUL, E., LIMPACH, C., PRATT, I., and WARFIELD, A., "Live migration of virtual machines," in *NSDI*, 2005.

[31] 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 *OSDI*, pp. 231–244, 2004.

[32] COHEN, I., ZHANG, S., GOLDSZMIDT, M., SYMONS, J., KELLY, T., and FOX, A., "Capturing, indexing, clustering, and retrieving system history," in *SOSP*, pp. 105–118, 2005.

[33] CORMODE, G. and GAROFALAKIS, M. N., "Sketching streams through the net: Distributed approximate query tracking," in *VLDB*, pp. 13–24, 2005.

[34] CORMODE, G., GAROFALAKIS, M. N., MUTHUKRISHNAN, S., and RASTOGI, R., “Holistic aggregates in a networked world: Distributed tracking of approximate quantiles,” in *SIGMOD Conference*, pp. 25–36, 2005.

[35] CUI, B., LU, H., XU, Q., CHEN, L., DAI, Y., and ZHOU, Y., “Parallel distributed processing of constrained skyline queries by filtering,” in *ICDE*, 2008.

[36] DEAN, J. and GHEMAWAT, S., “Mapreduce: Simplified data processing on large clusters,” in *OSDI*, pp. 137–150, 2004.

[37] DELIGIANNAKIS, A., STOUMPOS, V., KOTIDIS, Y., VASSALOS, V., and DELIS, A., “Outlier-aware data aggregation in sensor networks,” in *ICDE*, 2008.

[38] DHARMAPURIKAR, S., KRISHNAMURTHY, P., SPROULL, T. S., and LOCKWOOD, J. W., “Deep packet inspection using parallel bloom filters,” *IEEE Micro*, vol. 24, no. 1, pp. 52–61, 2004.

[39] DILMAN, M. and RAZ, D., “Efficient reactive monitoring,” in *INFOCOM*, 2001.

[40] DOULIGERIS, C. and MITROKOTSA, A., “Ddos attacks and defense mechanisms: classification and state-of-the-art,” *Computer Networks*, vol. 44, no. 5, pp. 643–666, 2004.

[41] DOYLE, R. P., CHASE, J. S., ASAD, O. M., JIN, W., and VAHDAT, A., “Model-based resource provisioning in a web service utility,” in *USENIX SITS*, 2003.

[42] ESTAN, C. and VARGHESE, G., “New directions in traffic measurement and accounting,” in *SIGCOMM02*.

[43] FOSTER, I. T., “The globus toolkit for grid computing,” in *CCGRID*, p. 2, 2001.

[44] GAO, L., WANG, M., and WANG, X. S., “Quality-driven evaluation of trigger conditions on streaming time series,” in *SAC*, 2005.

[45] GRIMMETT, G. and STIRZAKER, D., *Probability and Random Processes 3rd ed.* Oxford, 2001.

[46] GU, G., PERDISCI, R., ZHANG, J., and LEE, W., “Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection,” in *USENIX Security Symposium*, pp. 139–154, 2008.

[47] GUPTA, D., CHERKASOVA, L., GARDNER, R., and VAHDAT, A., “Enforcing performance isolation across virtual machines in xen,” in *Middleware*, pp. 342–362, 2006.

[48] HAYES, B., “Cloud computing,” *Commun. ACM*, vol. 51, no. 7, 2008.

[49] HUANG, L., GAROFALAKIS, M. N., JOSEPH, A. D., and TAFT, N., “Communication-efficient tracking of distributed cumulative triggers,” in *ICDCS*, p. 54, 2007.

[50] HUEBSCH, R., CHUN, B. N., HELLERSTEIN, J. M., LOO, B. T., MANIATIS, P., ROSCOE, T., SHENKER, S., STOICA, I., and YUMEREFENDI, A. R., "The architecture of pier: an internet-scale query processor," in *CIDR*, 2005.

[51] HUEBSCH, R., GAROFALAKIS, M. N., HELLERSTEIN, J. M., and STOICA, I., "Sharing aggregate computation for distributed queries," in *SIGMOD*, 2007.

[52] JAIN, A., HELLERSTEIN, J. M., RATNASAMY, S., and WETHERALL, D., "The case for distributed triggers," in *HotNets*, 2004.

[53] JAIN, N., AMINI, L., ANDRADE, H., KING, R., PARK, Y., SELO, P., and VENKATRAMANI, C., "Design, implementation, and evaluation of the linear road bnchmark on the stream processing core," in *SIGMOD*, 2006.

[54] JAIN, N., DAHLIN, M., ZHANG, Y., KIT, D., MAHAJAN, P., and YALAGANDULA, P., "Star: Self-tuning aggregation for scalable monitoring," in *VLDB*, pp. 962–973, 2007.

[55] JAIN, N., MAHAJAN, P., KIT, D., YALAGANDULA, P., DAHLIN, M., and ZHANG, Y., "Network imprecision: A new consistency metric for scalable monitoring," in *OSDI*, pp. 87–102, 2008.

[56] JAIN, N., YALAGANDULA, P., DAHLIN, M., and ZHANG, Y., "Self-tuning, bandwidth-aware monitoring for dynamic data streams," in *ICDE*, pp. 114–125, 2009.

[57] KASHYAP, S. R., RAMAMIRTHAM, J., RASTOGI, R., and SHUKLA, P., "Efficient constraint monitoring using adaptive thresholds," in *ICDE*, 2008.

[58] KASHYAP, S. R., TURAGA, D., and VENKATRAMANI, C., "Efficient trees for continuous monitoring," 2008.

[59] KEAHEY, K., FOSTER, I. T., FREEMAN, T., and ZHANG, X., "Virtual workspaces: Achieving quality of service and quality of life in the grid," *Scientific Programming*, 05.

[60] KERALAPURA, R., CORMODE, G., and RAMAMIRTHAM, J., "Communication-efficient distributed monitoring of thresholded counts," in *SIGMOD Conference*, pp. 289–300, 2006.

[61] KNUTH, D. E., *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd Ed.* Addison-Wesley, 1998.

[62] KO, S. and GUPTA, I., "Efficient on-demand operations in dynamic distributed infrastructures," in *LADIS*, 2008.

[63] KOSSMANN, D., "The state of the art in distributed query processing," *ACM Comput. Surv.*, vol. 32, no. 4, pp. 422–469, 2000.

[64] KRISHNAMURTHY, S., WU, C., and FRANKLIN, M. J., "On-the-fly sharing for streamed aggregation," in *SIGMOD Conference*, pp. 623–634, 2006.

[65] KRUEGEL, C. and VIGNA, G., "Anomaly detection of web-based attacks," in *CCS*, 2003.

[66] KUTARE, M., EISENHAUER, G., WANG, C., SCHWAN, K., TALWAR, V., and WOLF, M., "Monalytics: online monitoring and analytics for managing large scale data centers," in *ICAC*, pp. 141–150, 2010.

[67] LAGAR-CAVILLA, H. A., WHITNEY, J. A., SCANNELL, A. M., PATCHIN, P., RUMBLE, S. M., DE LARA, E., BRUDNO, M., and SATYANARAYANAN, M., "Snowflock: rapid virtual machine cloning for cloud computing," in *EuroSys*, 09.

[68] LI, J., MAIER, D., TUFTE, K., PAPADIMOS, V., and TUCKER, P. A., "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.

[69] LI, M., LIU, Y., and CHEN, L., "Non-threshold based event detection for 3d environment monitoring in sensor networks," in *ICDCS07*.

[70] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., and HONG, W., "Tinydb: an acquisitional query processing system for sensor networks," *ACM Trans. Database Syst.*, vol. 30, no. 1, pp. 122–173, 2005.

[71] MADDEN, S., FRANKLIN, M. J., HELLERSTEIN, J. M., and HONG, W., "Tag: A tiny aggregation service for ad-hoc sensor networks," in *OSDI*, 2002.

[72] MADDEN, S., SHAH, M. A., HELLERSTEIN, J. M., and RAMAN, V., "Continuously adaptive continuous queries over streams," in *SIGMOD*, 2002.

[73] MANJHI, A., NATH, S., and GIBBONS, P. B., "Tributaries and deltas: Efficient and robust aggregation in sensor network streams," in *SIGMOD*, 2005.

[74] MARZULLO, K. and WOOD, M. D., "Tools for constructing distributed reactive systems," 1991.

[75] MCNETT, M., GUPTA, D., VAHDAT, A., and VOELKER, G. M., "Usher: An extensible framework for managing clusters of virtual machines," in *LISA*, pp. 167–181, 2007.

[76] MENG, S., IYENGAR, A. K., ROUVELLOU, I. M., and LIU, L., "Violation likelihood based state monitoring." Technical Report, 2011.

[77] MENG, S., IYENGAR, A. K., ROUVELLOU, I. M., and LIU, L., "Cloud prism: Prediction-based application provisioning service." Technical Report, 2012.

[78] MENG, S., KASHYAP, S. R., VENKATRAMANI, C., and LIU, L., "Resource-aware application state monitoring," *IEEE Transactions on Parallel and Distributed Systems*, p. to appear.

[79] MENG, S., KASHYAP, S. R., VENKATRAMANI, C., and LIU, L., "Remo: Resource-aware application state monitoring for large-scale distributed systems," in *ICDCS*, pp. 248–255, 2009.

[80] MENG, S. and LIU, L., "Reliable state monitoring in large-scale distributed systems." Technical Report, 2011.

[81] MENG, S., LIU, L., and SOUNDARARAJAN, V., "Tide: Achieving self-scaling in virtualized datacenter management middleware," in *Middleware'10*.

[82] MENG, S., LIU, L., and WANG, T., "State monitoring in cloud datacenters," *IEEE Trans. Knowl. Data Eng.*, vol. 23, no. 9, pp. 1328–1344, 2011.

[83] MENG, S., WANG, T., and LIU, L., "Monitoring continuous state violation in datacenters: Exploring the time dimension," in *ICDE*, pp. 968–979, 2010.

[84] NARAYANAN, D., DONNELLY, A., THERESKA, E., ELNIKETY, S., and ROWSTRON, A. I. T., "Everest: Scaling down peak loads through i/o off-loading," in *OSDI*, pp. 15–28, 2008.

[85] NURMI, D., WOLSKI, R., GRZEGORCZYK, C., OBERTELLI, G., SOMAN, S., YOUSEFF, L., and ZAGORODNOV, D., "The eucalyptus open-source cloud-computing system," in *CCGRID*, 2009.

[86] OLSTON, C., JIANG, J., and WIDOM, J., "Adaptive filters for continuous queries over distributed data streams," in *SIGMOD Conference*, 2003.

[87] OLSTON, C., LOO, B. T., and WIDOM, J., "Adaptive precision setting for cached approximate values," in *SIGMOD*, 2001.

[88] OLSTON, C. and WIDOM, J., "Offering a precision-performance tradeoff for aggregation queries over replicated data," in *VLDB*, 2000.

[89] PADALA, P., SHIN, K. G., ZHU, X., UYSAL, M., WANG, Z., SINGHAL, S., MERCHANT, A., and SALEM, K., "Adaptive control of virtualized resources in utility computing environments," in *EuroSys07*.

[90] PARK, K. and PAI, V. S., "Comon: a mostly-scalable monitoring system for planetlab," *Operating Systems Review*, vol. 40, no. 1, pp. 65–74, 2006.

[91] PETERSON, L. L., "Planetlab: Evolution vs. intelligent design in planetary-scale infrastructure," in *USENIX ATC*, 2006.

[92] PROJECT, T. N., "Ntp faq." `http://www.ntp.org/ntpfaq/NTP-s-algo.htm#Q-ACCURATE-CLOCK`.

[93] PU, X., LIU, L., MEI, Y., SIVATHANU, S., KOH, Y., and PU, C., "Understanding performance interference of i/o workload in virtualized cloud environments," in *IEEE Cloud*, 2010.

[94] RAGHAVAN, B., VISHWANATH, K. V., RAMABHADRAN, S., YOCUM, K., and SNOEREN, A. C., "Cloud control with distributed rate limiting," in *SIGCOMM07*.

[95] RASCHID, L., WEN, H.-F., GAL, A., and ZADOROZHNY, V., "Monitoring the performance of wide area applications using latency profiles," in *WWW03*.

[96] SEKAR, V., REITER, M. K., WILLINGER, W., ZHANG, H., KOMPELLA, R. R., and ANDERSEN, D. G., "csamp: A system for network-wide flow monitoring," in *NSDI*, pp. 233–246, 2008.

[97] SHARFMAN, I., SCHUSTER, A., and KEREN, D., "A geometric approach to monitoring threshold functions over distributed data streams," in *SIGMOD Conference*, pp. 301–312, 2006.

[98] SILBERSTEIN, A., BRAYNARD, R., and YANG, J., "Constraint chaining: on energy-efficient continuous monitoring in sensor networks," in *SIGMOD*, 2006.

[99] SILBERSTEIN, A., MUNAGALA, K., and YANG, J., "Energy-efficient monitoring of extreme values in sensor networks," in *SIGMOD*, 2006.

[100] SILBERSTEIN, A. and YANG, J., "Many-to-many aggregation for sensor networks," in *ICDE*, pp. 986–995, 2007.

[101] SOUNDARARAJAN, V. and ANDERSON, J. M., "The impact of management operations on the virtualized datacenter," in *ISCA*, 2010.

[102] SRIVASTAVA, U., MUNAGALA, K., and WIDOM, J., "Operator placement for in-network stream query processing," in *PODS*, pp. 250–258, 2005.

[103] STEWART, C., KELLY, T., and ZHANG, A., "Exploiting nonstationarity for performance prediction," in *EuroSys*, pp. 31–44, 2007.

[104] SÜLI, E. and MAYERS, D. F., *An Introduction to Numerical Analysis*. Cambridge University Press, '03.

[105] SUTHERLAND, T. M., LIU, B., JBANTOVA, M., and RUNDENSTEINER, E. A., "D-cape: distributed and self-tuned continuous query processing," in *CIKM*, pp. 217–218, 2005.

[106] TERREMARK, "vcloud express." http://vcloudexpress.terremark.com/.

[107] TURAGA, D. S., VLACHOS, M., VERSCHEURE, O., PARTHASARATHY, S., FAN, W., NORFLEET, A., and REDBURN, R., "Yieldmonitor: Real-time monitoring and predictive analysis of chip manufacturing data," 2008.

[108] URGAONKAR, B., PACIFICI, G., SHENOY, P. J., SPREITZER, M., and TANTAWI, A. N., "An analytical model for multi-tier internet services and its applications," in *SIGMETRICS*, pp. 291–302, 2005.

[109] VISWANATHAN, K., CHOUDUR, L., TALWAR, V., WANG, C., MACDONALD, G., and SATTERFIELD, W., "Ranking anomalies in distributed systems," in *NOMS*, 2012.

[110] VMWARE, "Distributed resource scheduling and distributed power management." `http://www.vmware.com/products/drs/`.

[111] VMWARE, "Linked-vsphere server." `http://www.vmware.com/support/vsphere4`.

[112] VMWARE, "vMotion." `http://www.vmware.com/products/vmotion`.

[113] VMWARE, "VMware HA." `http://www.vmware.com/products/high-availability/`.

[114] VMWARE, "VMware Server Consolidation." `http://www.vmware.com/solutions/consolidation/`.

[115] VMWARE, "VMware View." `http://www.vmware.com/products/view/`.

[116] VMWARE, "vSphere." `http://www.vmware.com/products/vsphere/`.

[117] WANG, C., "Ebat: Online methods for detecting utility cloud anomalies," in *MDS*, 2009.

[118] WANG, C., SCHWAN, K., TALWAR, V., EISENHAUER, G., HU, L., and WOLF, M., "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *ICAC*, pp. 141–150, 2011.

[119] WANG, C., TALWAR, V., SCHWAN, K., and RANGANATHAN, P., "Online detection of utility cloud anomalies using metric distributions," in *NOMS*, pp. 96–103, 2010.

[120] WHITE, B., LEPREAU, J., STOLLER, L., RICCI, R., GURUPRASAD, S., NEWBOLD, M., HIBLER, M., BARB, C., and JOGLEKAR, A., "An integrated experimental environment for distributed systems and networks," in *OSDI*, 2002.

[121] WIKIPEDIA, "Mutual information." `http://en.wikipedia.org/wiki/Mutual_information`.

[122] XIANG, S., LIM, H.-B., TAN, K.-L., and ZHOU, Y., "Two-tier multiple query optimization for sensor networks," in *ICDCS*, p. 39, 2007.

[123] YALAGANDULA, P. and DAHLIN, M., "A scalable distributed information management system," in *SIGCOMM*, pp. 379–390, 2004.

[124] YANG, X., LIM, H.-B., ÖZSU, M. T., and TAN, K.-L., "In-network execution of monitoring queries in sensor networks," in *SIGMOD*, 2007.

[125] Zhang, R., Koudas, N., Ooi, B. C., and Srivastava, D., “Multiple aggregations over data streams,” in *SIGMOD*, 2005.

[126] Zhao, Y., Tan, Y., Gong, Z., Gu, X., and Wamboldt, M., “Self-correlating predictive information tracking for large-scale production systems,” in *ICAC*, pp. 33–42, 2009.

[127] Zheng, W., Bianchini, R., Janakiraman, G. J., Santos, J. R., and Turner, Y., “Justrunit: Experiment-based management of virtualized data centers,” in *USENIX ATC*, 2007.

[128] Zhou, Y., Chakrabarty, D., and Lukose, R. M., “Budget constrained bidding in keyword auctions and online knapsack problems,” in *WWW*, pp. 1243–1244, 2008.

[129] Zipf, G. K., *Human Behavior and the Principle of Least Effort*. Addison-Wesley, 1949.

VITA

Shicong Meng was born and raised in Wuhu, a beautiful city on the south bank of Changjiang river in east China. He received a Bachelor of Science degree from East China Normal University, Shanghai, China in 2004, and a Master of Science degree from Shanghai Jiaotong University in 2007. Subsequently, he moved to Atlanta to pursue a Ph.D. in Computer Science at the College of Computing at Georgia Institute of Technology. As a member of the DiSL research group and CERCS at the College of Computing, he conducted research on various aspects of distributed data intensive systems under the guidance of Prof. Ling Liu. His research in these projects has resulted in numerous publications that have appeared in various international conferences and journals on distributed systems and data management. He has also been a collaborator with the IBM T.J. Watson Research Center and VMware. He received an IBM Ph.D. Fellowship in 2011 and holds or applied for a number of patents on his work at both VMware and IBM, dealing with Cloud Datacenter Monitoring and Management.

Monitoring-as-a-Service in the Cloud

Shicong Meng

241 Pages

Directed by Professor Ling Liu

State monitoring is a fundamental building block for Cloud services. The demand for providing state monitoring as services (MaaS) continues to grow and is evidenced by CloudWatch from Amazon EC2, which allows cloud consumers to pay for monitoring a selection of performance metrics with coarse-grained periodical sampling of runtime states. One of the key challenges for wide deployment of MaaS is to provide better balance among a set of critical quality and performance parameters, such as accuracy, cost, scalability and customizability.

This dissertation research is dedicated to innovative research and development of an elastic framework for providing state monitoring as a service (MaaS). We analyze limitations of existing techniques, systematically identify the need and the challenges at different layers of a Cloud monitoring service platform, and develop a suite of distributed monitoring techniques to support for flexible monitoring infrastructure, cost-effective state monitoring and monitoring-enhanced Cloud management. At the monitoring infrastructure layer, we develop techniques to support multi-tenancy of monitoring services by exploring cost sharing between monitoring tasks and safeguarding monitoring resource usage. To provide elasticity in monitoring, we propose techniques to allow the monitoring infrastructure to self-scale with monitoring demand. At the cost-effective state monitoring layer, we devise several new state monitoring functionalities to meet unique functional requirements in Cloud monitoring. Violation likelihood state monitoring explores the benefits of consolidating monitoring workloads by allowing utility-driven monitoring intensity tuning on individual monitoring tasks and identifying correlations between monitoring tasks.

Window based state monitoring leverages distributed windows for the best monitoring accuracy and communication efficiency. Reliable state monitoring is robust to both transient and long-lasting communication issues caused by component failures or cross-VM performance interferences. At the monitoring-enhanced Cloud management layer, we devise a novel technique to learn about the performance characteristics of both Cloud infrastructure and Cloud applications from cumulative performance monitoring data to increase the cloud deployment efficiency.