Enhanced Monitoring-as-a-Service for Effective Cloud Management

Shicong Meng, *Student Member, IEEE,* and Ling Liu, *Senior Member, IEEE*

Abstract—This paper introduces the concept of monitoring-as-a-service (MaaS), its main components, and a suite of key functional requirements of MaaS in Cloud. We argue that MaaS should support not only the conventional state monitoring capabilities, such as instantaneous violation detection, periodical state monitoring and single tenant monitoring, but also performance-enhanced functionalities that can optimize on monitoring cost, scalability, and the effectiveness of monitoring service consolidation and isolation. In this paper we present three enhanced MaaS capabilities and show that window based state monitoring is not only more resilient to noises and outliers, but also saves considerable communication cost. Similarly, violation-likelihood based state monitoring can dynamically adjust monitoring intensity based on the likelihood of detecting important events, leading to significant gain in monitoring service consolidation. Finally, multi-tenancy support in state monitoring allows multiple Cloud users to enjoy MaaS with improved performance and efficiency at more affordable cost. We perform extensive experiments in an emulated Cloud environment with real world system and network traces. The experimental results suggest that our MaaS framework achieves significant lower monitoring cost, higher scalability and better multi-tenancy performance.

Index Terms—Cloud Monitoring, Monitoring Service, State Monitoring, Violation Likelihood, Multi-Tenancy

1 Introduction

Cloud systems and applications often run over a large number of commodity computing nodes. Web applications are hosted over distributed web servers, application servers and databases to handle the sheer amount of workload [1]. Systems such as Bigtable and Hadoop are deployed over hundreds, even thousands, of machines to achieve high scalability and reliability. As a result, state monitoring is a critical and indispensable component of Cloud management for mission critical tasks, such as safeguarding performance [2], detecting attacks [3], datacenter-wide profiling [4] and masking the underlying component failures in massively distributed computing environments [5].

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.

In this paper we present a model of monitoring-as-a-service (MaaS) by introducing the main functional and non-functional requirements of MaaS and a coordinator-based multi-node monitoring framework. Although most Cloud datacenters often use instantaneous state monitoring, periodical sampling and single-tenant monitoring as basic monitoring functionalities, we argue that MaaS should provide window-based monitoring in addition to the instantaneous state monitoring, violation-likelihood based state collection in addition to the period sampling, and multi-tenant state monitoring in addition to making monitoring solely as a single-tenant service. We also identify three fundamental requirements for MaaS: minimizing the communication overhead, minimizing noise, and maximizing the monitoring effectiveness. We show that the new functionalities meet these requirements and can be implemented under a distributed coordinator-based monitoring framework.

Concretely, at the global state violation detection level, the window based monitoring not only provides the flexibility in state violation detection, but also saves significant communication cost and reduces monitoring noise, compared with the instantaneous state monitoring. Similarly, at the local state monitoring level, the violation-likelihood based sampling, as an alternative, offers the option and flexibility to dynamically adjust monitoring intensity based on how likely a state violation will be detected. By carefully controlling the violation likelihood estimate, we can save considerable monitoring cost at the price of negligible monitoring error. Furthermore, we show the benefit of provisioning MaaS in a multi-tenant platform by developing an adaptive planning technique that safeguards the per-node monitoring resource usage for concurrent running tasks, and at the same time, optimizes the underlying communication topology for efficiency and scalability.

We evaluate the effectiveness of our MaaS approach with extensive experiments based on both a real world monitoring setup and synthetic ones (for extreme scales or workloads). The experimental results suggest that our multi-level MaaS approach can deliver state monitoring services through more flexible interfaces, with significantly lower cost (50%-90% communication cost reduction, 10%-90% sampling cost reduction), higher scalability and better isolation between tasks.

To the best of our knowledge, this is the first systematic approach to develop a suite of enhanced techniques for

• *S. Meng and L. Liu are with the College of Computing, Georgia Institute of Technology, Atlanta, GA, 30332.*
E-mail: {smeng, lingliu}@cc.gatech.edu

monitoring-as-a-service in Cloud environments. This paper makes three unique contributions. First, we describe a basic model of monitoring-as-a-service, including functional, non-functional requirements and a coordinator-based implementation framework. Second, we develop a suite of three advanced state monitoring techniques: the window-based monitoring, the violation-likelihood sampling and the multi-tenant monitoring. We analyze the efficiency, accuracy and scalability challenges in delivering these monitoring functions as Cloud services. Finally, we conduct extensive experimental evaluation to show the effectiveness of our approach to MaaS provisioning in the Cloud.

The rest of the paper is organized as follows. Section 2 provides an overview of our MaaS framework. We present details of the window-based violation detection algorithm in Section 3. In Section 4, we introduce the violation likelihood based local state monitoring. Section 5 presents our multi-tenancy enabling monitoring planning techniques. In Section 6, we present our experimental results. We discuss related work in Section 7. Section 8 concludes this paper and summaries our future research directions.

2 MaaS Overview

We consider Cloud service providers that offer Infrastructure-as-a-Service (IaaS) and Platform-as-a-Service (PaaS) as the targeted monitoring service providers. IaaS delivers computing infrastructure, typically a platform virtualization environment, as a service, along with raw (block) storage and networking. PaaS provides a higher level computing platform as a service, which often facilitates the deployment of applications without the cost and complexity of buying and managing the underlying hardware, OS, middleware. Monitoring-as-a-Service (MaaS) is a general framework for realizing monitoring functionalities in the Cloud. We focus on the widely used online state monitoring that continuously watches a certain state of a monitored object (e.g., application/system/network) in this paper.

2.1 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 [6] enables users to monitor the overall timeout requests on a web application deployed over multiple server instances. Users can receive a state alert when the overall timeout requests exceed a threshold defined by an SLA, and users can dynamically provision new server instances to the web application to improve performance.

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 timeout requests, deviates from a normal state. State monitoring is widely used in Cloud management scenarios such as traffic engineering [7], QoS assurance [2], fighting DoS attack [5]. It is also frequently used for Cloud user monitoring tasks such as Cloud application auto-scaling [8], distributed rate limiting of Cloud services [1], application availability monitoring [9], Cloud service auditing [9], monitoring-based application deployment optimization [10], etc.

State monitoring is also an ideal monitoring form for being offered as services due to its flexibility in meeting users' monitoring needs. For instance, states can be easily defined over common performance metrics (e.g., CPU utilization), or metrics defined by users (e.g., application-level cache hit rate). Users can even provide scripts that generate application state values so that state monitoring services can invoke such scripts during sampling operations[1].

2.2 Basic and Advanced Functionality in MaaS

Provisioning state monitoring services requires at least three core functionalities: global violation detection, local state monitoring and multi-tenancy support. In this section we define each functionality and show how basic and advanced state monitoring techniques can be used and developed to improve the overall efficiency and scalability of state monitoring. We would like to note that we promote the monitoring-as-a-service as a metaphor. What comprises of the suite of mandatory functionalities of MaaS is yet to be studied and standardized through threads of research and development efforts. We conjecture that the set of techniques described in this paper should be an integral part of MaaS for scalable and effective Cloud state management.

One monitoring example is distributed rate limiting [1]. Cloud services such as the Amazon's S3 storage service are hosted by a large number of servers distributed around the globe. Although users can access such a service at any hosting server, the aggregated access rate of a user is often limited to a level (defined by the service level agreement). To enforce such distributed rate limiting, one needs to monitor service access rates of a user at all distributed hosting servers, and check whether the aggregated access rate exceeds the level associated with this user. We next show how such Cloud monitoring tasks introduce challenges to global violation detection, local state collection and multi-tenancy support.

Global Violation Detection refers to the process of deciding whether to trigger global state violations or not based on monitoring values collected from distributed monitors. For the distributed rate limiting monitoring example, suppose that we want to be alerted by a state violation when the total access rate of a user exceeds a given level. To detect such global state violations, one needs to collect and aggregate the local access rates, i.e., the access rate of the user on individual hosting servers. Since the monitoring results determine how a user can access a Cloud service, ensuring monitoring accuracy is clearly important. In addition, as a distributed rate limiting task may involve aggregating local states from hundreds, or even thousands, of Cloud service hosting servers which may be located in geographically distributed datacenters (possibly on different continents), minimizing the communication cost is also critical [1].

The basic and most commonly used state violation detection model is the instantaneous state violation model which triggers a global violation whenever the monitored value exceeds a given threshold [11], [12], [13]. Unfortunately, this model may cause frequent and unnecessary violations in the presence of sudden surges, noises and outliers in the monitored values. It can even cause unnecessary counter-measures (e.g., denial of user access due to a transient increase of the access rate). In many monitoring scenarios, users often prefer receiving alerts about established state violations and minimizing false positives, especially in scenarios like distributed rate limiting where false positives may hurt user experiences. The window based violation detection model reports only violations that are continuous within an adjustable time window. It not only provides more flexibility in monitoring sensitivity and higher

1. Such scripts should meet certain performance requirements (e.g., per-sampling CPU consumption and execution time) before used in production

accuracy, but also allows violation detection algorithms to leverage distributed temporal monitoring windows for better communication efficiency. We argue that in the context of Cloud, MaaS should support both the instantaneous and the window based state monitoring.

Local State Monitoring is the process of collecting monitored state data on individual monitoring nodes. In the distributed rate limiting example, local state monitoring may involve expensive operations such as scanning recent access logs on a server to obtain the monitored access rate statistics. The collected monitoring data are then used to detect global state violations. Local state monitoring typically uses the periodic sampling (probing) approach to collect local states at monitoring nodes. Although periodic sampling is widely used due to its simplicity, it may not be the most suitable method for all types of local state violation detection needs due to its context blind limitations. For example, periodic sampling can be wasteful when the access rate of a user is both trivial and relatively stable (i.e., unlikely to introduce state violations), or when the monitoring needs change over time (e.g., tolerating more access rate violations during off-peak hours).

We argue that the MaaS model should provide the periodic sampling as the basic functionality, and offer violation likelihood based sampling as an alternative technique. The violation likelihood sampling method can dynamically adjust sampling intensity based on the measured likelihood of detecting violations. It saves significant sampling cost when monitored states are stable and violations are rare events. Violation likelihood based sampling techniques fit nicely with the monitoring service consolidation principle as one application may not need intensive monitoring constantly and different monitoring tasks are not likely to perform intensive sampling all the time.

Multi-Tenancy Support. Though single tenant based state monitoring service is the basic functionality of MaaS, we argue that multi-tenancy support is an important addition to MaaS. For the distributed rate limiting example, because monitoring is performed on a per-user basis, a single Cloud service may involve thousands, even millions, of monitoring tasks. For datacenters running many Cloud services, the total number of monitoring tasks can be tremendous. Since these tasks are very likely to overlap in terms of servers and metrics being monitored (e.g., two monitored services may run on a similar set of servers), multi-tenant MaaS offers unique opportunities for improving monitoring efficiency and scalability through cost-sharing and isolation. With the multi-tenancy support, one can optimize the underlying communication topology between monitor nodes, de-duplicate monitoring workloads, and provide better matching of the monitoring workloads with node-level monitoring capacities while ensuring adequate performance isolation between concurrent monitoring tasks.

Our Goal. MaaS is an emerging paradigm of Cloud management and it involves many important research problems such as user API, interfaces, QoS and pricing model which all deserve comprehensive studies. In this paper, by developing new techniques at the levels of global violation detection, local state monitoring and multi-tenancy support, we intend to demonstrate that a layered approach has the potential to meet the highly diverse and unique Cloud monitoring needs. Although techniques presented in this paper cannot entirely capture the diverse monitoring needs from different users, it is possible to extend this layered approach by enhancing techniques at one level or developing techniques at a new level to address emerging monitoring needs. The subsequent three sections provide technical details on the window-based monitoring, the violation likelihood based sampling and the multi-tenancy support in MaaS.

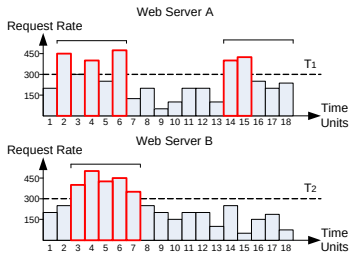


Fig. 1: Implications on Communication Cost

3 Detecting Window-based Violation

In this section we present a novel distributed implementation of the window-based detection model which achieves significant communication cost savings over existing techniques. We also introduce monitoring parameter tuning techniques that achieve the optimal communication efficiency.

3.1 Implication of Detection Models

The window-based detection model triggers state alerts only when the monitored value exceeds a threshold *continuously* for at least L time units. By changing the size of monitoring window L, users can adjust the sensitivity of violation detection. While the semantic differences between the instantaneous and the window-based models are straightforward, their implication on the communication cost is not immediately clear. In fact, the window-based model, when implemented properly with a distributed coordinator-based framework, saves significant communication cost. We next reveal the communication cost implication of detection models through a concrete example.

Instantaneous Detection. Figure 1 shows a snippet of HTTP request rate traces collected from two web servers in a geographically distributed server farm [14], where time is slotted into 5-second units. Let us first consider a state monitoring task which triggers a *global violation* when the sum of request rates at two servers exceeds $T = 600$. Let $v_i(t)$ be the monitored value on monitor i at time t, and n to be the number of monitor nodes in the task. In addition, nodes involved in distributed state monitoring 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) [15].

Distributed state monitoring could introduce significant communication cost. For the distributed rate limiting example, monitoring 1k+ Cloud service access flows distributed over 500 servers can contribute up to 230MB/s bandwidth consumption just for monitoring [1] with simple collecting and aggregation based monitoring techniques. Note that much of the communication are long-distant as servers may be located in geographically distributed datacenters. In addition, the monitoring message processing cost such as transmission or protocol overhead can be substantial even when the contained monitoring data is trivial. For instance, a typical monitoring message delivered via TCP/IP protocol has a message header of at least 100 bytes with service related information, while an integer monitoring data is just 4 bytes. We will present a detailed per-message overhead analysis in Section 5.1.

With the instantaneous detection model, the line of existing works tries to minimize the number of monitoring messages by

decomposing 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 a *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 $\sum_{i=1}^{n} v_i(t) \leqslant T$. Note that a global poll is necessary for determining whether $\sum_{i=1}^{n} v_i(t) \leqslant T$ because $\sum_{i=1}^{n} v_i(t) \leqslant \sum_{i=1}^{n} T_i$ may not hold any more.

For simplicity, we assume server A and B have the same local thresholds $T_1 = T_2 = T/2 = 300$, as indicated by the dashed lines. A local violation happens when a bar raises above a dashed line, as indicated by bars with borders. Correspondingly, server A and B report local violations respectively at time unit 2,4,6,14,15, and time unit 3-7 (10 messages). When receiving local violation reports, the coordinator invokes global polls at time unit 2,3,5,7,14,15 ($6 = 12$ messages) to collect values from servers that do not report local violations. Thus, the total message number is $10 + 12 = 22$.

Centralized Window-based Detection. Now we perform the window-based state monitoring to determine whether there exist continuous global violations against T lasting for $L = 8$ time units. We start with the most intuitive centralized approach, where everything is the same as the above example except that the coordinator triggers state alerts only when observing continuous global violations of 8 time units. As a result, the communication cost is the same as before, 22 messages.

Distributed Window-based Detection. Invoking a global poll for every local violation is not necessary. Since only continuous global violations count, the coordinator can delay global polls unless it observes 8 continuous time units with local violations. When it observes a time unit t with no local violation, it can discard all pending global polls, as the violation is not continuous. This modified scheme avoids all 6 global polls which reduces the total message number to 10.

We can further reduce the monitor-side communication cost by filtering continuous local violation reports. For instance, assume both servers use 5-time-unit filtering windows. Server A reports a local violation at time unit 2, and then enters a filtering window, during which it does not 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 a 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 violations 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.

Insights and Challenges. The above example suggests that a good distributed implementation of the window-based model may achieve great communication saving. Developing a complete approach, however, still involves several challenges. First, we should devise a distributed detection algorithm that not only ensures the correctness of monitoring, but also utilizes the monitoring window to minimize communication cost. Second, given that the distributed detection algorithm ensures the monitoring correctness, we should also develop techniques that automatically tune monitoring parameters (e.g., local threshold, local filtering window size) to always maintain low communication cost in different conditions. We next address these challenges in the rest of this section.

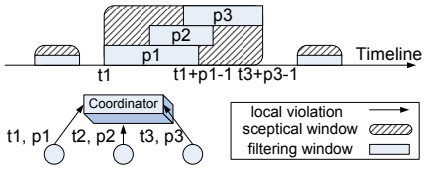


Fig. 2: Filtering Windows and Skeptical Windows.

3.2 The Distributed Detection Algorithm

The detection algorithm consists of two parts, the monitor node side algorithm and the coordinator side algorithm:

The Monitor Node Side. A monitor node i employs 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 length of a time window and is defined over $[0, L]$. Specifically, filtering windows are defined as follows.

Definition 1: 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 $t_s(\tau)$ and $t_e(\tau)$ to denote the start time and the end time of τ, and $|\tau|$ to represent the remaining length of a filtering window τ with regarding to t, i.e., $|\tau| = t_e(\tau) - t$. 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 , it sends a local violation report to the coordinator, and then suppresses all local violation reports for p_i time units. It starts to detect and report violations again only after p_i time units. 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.

The Coordinator Side. The coordinator side algorithm "reassembles" potential periods of local violations indicated by local violation reports into a potential period of continuous global violations, 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 a reported local violation from node i as continuous local violations lasting p_i time units, i.e., assuming filtering windows fully filled with local violations. It concatenates reported filtering windows that overlap in time into the skeptical window which is defined as follows:

Definition 2: A *skeptical window* κ is a period of time consisting of most recent overlapped filtering windows related with reported local violations since the previous global poll. Initially, the size of a skeptical window $|\kappa|$ is 0 and its start and end time are set to the time when the monitoring algorithm starts. Given a set of filtering windows $\mathbb{T} = \{\tau_i | i \in [1, n]\}$ observed at time t, κ can be updated as follows:

$$t_s(\kappa') \leftarrow \begin{cases} t_s(\kappa) & t_e(\kappa) \geq t \\ t & otherwise \end{cases} \quad (1)$$

$$t_e(\kappa') \leftarrow \begin{cases} t + \max\limits_{\forall \tau_i \in \mathbb{T}} \{t_e(\kappa) - t, |\tau_i|\} & t_e(\kappa) \geq t \\ \max\limits_{\forall \tau_i \in \mathbb{T}} \{t, t_e(\tau_i)\} & otherwise \end{cases} \quad (2)$$

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, $|\kappa| = t_e(\kappa) - t_s(\kappa) + 1$. Essentially, Equation 1 indicates that the beginning of the skeptical window is the beginning of ongoing, continuous local violations, and Equation 2 shows that the end of the skeptical window is the end of ongoing, continuous local violations. In our motivating example, server A and B with $p_A = p_B = 5$ report local violations 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 2 shows an illustrative example of skeptical windows.

Recall that L is the length of the monitoring window and a state alert occurs only when L continuous global violations exist. When $t - t_s(\kappa) = L$, it indicates that there may exist continuous local violations for the last L time units (which could lead to continuous global violations of L time units). 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 $v_i(t-j), \forall j \in [0, t']$, 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 violations also provide past v_i values. After a global poll, if the coordinator detects continuous global violations 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 the received v_i. Clearly, the computation cost of both monitor and coordinator algorithms is trivial.

3.2.1 Correctness and Efficiency

The detection 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 violations. Second, the coordinator never triggers false state alerts (false positive) as it triggers state alerts only after examining complete local violation information. In addition, through analysis, we find that the distributed detection algorithm reduces communication cost by approximately 66% for a typical configuration, and the saving grows with the window size and the number of participating nodes. Our experimental results suggest that the actual gain is generally better (50% to 90% reduction in communication cost) with parameter tuning. We refer readers to [16] for details of the correctness proof and efficiency analysis.

3.2.2 Parameter Tuning

The performance of the violation detection algorithm also depends on the setting of local monitoring parameters, i.e., T_i and p_i. As a simple example, increasing the local threshold T_i or the filtering window size p_i of node i reduces the amount of local violation reports on node i. However, it is not immediately clear how local monitoring parameters should be set to minimize monitoring cost. For instance, while increasing p_i reduces local violation reports, the size of reported local violation periods also increases (same size as p_i). Consequently, reported violation periods from different nodes are more likely to overlap with each other. This causes the skeptical window to grow quickly, which in turn leads to frequent triggering of expensive global polls ($2n$ messages where n is the number of nodes).

Cost Analysis. To find the most communication efficient parameters, we build a communication cost model that links the parameters to the expected communication cost of the algorithm. Specifically, the communication cost model has the following form, $C = \sum_{i=1}^{n} C_l P_l(i) + \sum_{j=L}^{\infty} C_g^j P_g^j$, where C_l and $P_l(i)$ denote the communication cost of sending a local violation report and the probability of sending it at one time unit on node i respectively, C_g^j represents the cost associated with a global poll which collects values for the past j time slots, P_g^j is the probability of observing such a global poll. The model produces the expected cost of both local violation reporting and global polls for a given set of parameters and the observed monitored value distribution.

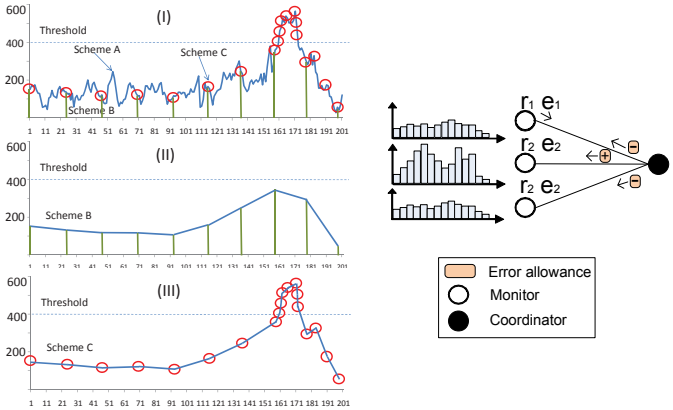
The Tuning Scheme. Based on the cost model, we develop a parameter tuning scheme to determine the best parameters for different monitoring tasks and environments. The scheme performs an EM-style local search process which iteratively looks for values leading to less cost. It starts with two sets of initial values for T_i and p_i. Iteratively, the scheme fixes one set of parameters and performs hill climbing to optimize the other set of parameters until reaching a local minimum. This scheme is computationally lightweight, and can be invoked periodically to catch up environment changes (e.g., monitored value distributed changes). On a Intel Core 2 Duo 2.26MHz processor, the CPU consumption of the parameter tuning scheme contributed is negligible in all our experiments. Due to space limitation, we refer readers to [16] for details of our parameter tuning techniques.

The scalability of our approach is primarily determined by the capability of the coordinator. While we do not observe scalability issues for tasks with up to 1k nodes, it is possible to support extra-scale (>1k) monitoring with a multi-level tree where the leaf nodes are the monitors and all the other nodes are coordinators. In this case, our detection algorithm can be invoked in a recursive manner from leaf monitors to the root coordinator. Node failures and disconnection can be quite common for large-scale distributed monitoring. One possible technique to prevent such issues is to use heartbeat messages. If the coordinator does not receive heartbeats from a certain node for a while, it considers the node as unavailable and notifies the user that the current monitoring result may not be reliable. The coordinator node, while described as a single logical entity, can also be implemented with master-secondary instances for high-availability. We refer readers to [17] for advanced techniques in extra-scale and robust state monitoring.

4 Violation-Likelihood Based Sampling

Most existing monitoring systems adopt periodical sampling, where a monitoring node periodically collects local state information with a fixed waiting interval. Despite its simplicity, periodical sampling has its limitation in delivering cost-effective state monitoring services. Using network traffic logs to detect unusual network usage is common in many online network management scenarios. For example, Distributed Heavy Hitter (DHH) [7] detection finds IP addresses that account for high volume traffic, e.g., IPs receiving more than 5k packets in the last minute. To perform a DHH monitoring tasks, traffic logs (e.g., netflow or tcpdump), are first periodically collected from network devices such as routers, switches, firewalls and gateways to a monitoring system. The monitoring system then scans the collected logs to find heavy hitters. For brevity, we refer each cycle of log collection and processing as sampling.

Chart (I) in Figure 3(a) shows an example trace of packets received by an IP in a DHH detection task (observed on a monitor). For the sake of the discussion, we assume the



(a) A Motivating Example (b) Distributed Coordination

Fig. 3: Violation Likelihood Based Sampling

monitor uses the instantaneous detection model, and it reports a local violation when the packet number exceeds the threshold (dashed line). The trace is produced by a scheme with frequent periodical sampling which we refer to as scheme A. Scheme A records details of the value changes and can detect the local violation in the later portion of the trace. Nevertheless, the high sampling frequency also introduces high monitoring cost. Traffic logs can be quite large for networks with moderate or high volume traffic. A production router may export netflow logs containing millions of flows in a 1-minute interval, which corresponds to a plain text log file with size close to 100MB. Other fine-grained logs such as tcpdump are even more space-consuming. The sampling (i.e., transmission and processing) cost of such logs is considerable. In addition, tasks such as DHH detection often require simultaneous collecting of traffic logs from multiple network devices to obtain the global traffic information. Hence, the total sampling cost grows linearly with the size of the network.

As the frequent sampling of scheme A does not yield useful state violation information for most part of the trace, one may suggest using a lower sampling frequency to reduce monitoring cost. For example, scheme B reduces considerable monitoring cost by using a relatively large monitoring interval (each bar denotes a sampling operation). The resulting trace shown in Chart (II), however, fails to detect the local violation (between the gap of two continuous samples of scheme B). Scheme A and B suggest one limitation of periodical sampling, a stiff tradeoff of cost and accuracy, which provides poor flexibility in achieving monitoring cost-effectiveness.

One observation worth noting is that the traffic peak causing the state violation does not come in a sudden. Before it arrives, the traffic level gradually grows towards the threshold and the possibility of violation increases. Based on this observation, an alternative scheme may adjust sampling intervals on the fly based on the likelihood of detecting violations, which we refer to violation likelihood. Scheme C demonstrates an example. As the trace in Chart (III) shows, scheme C uses a low sampling frequency at first, then switches to high frequency sampling when the traffic level increases and moves towards the threshold. This violation-likelihood based sampling can be a useful alternative for providing flexible and efficient Cloud monitoring services[2]. Because most state violation tasks experience high violation likelihood only during a small portion of their lifetime (e.g., a few peak hours during a day), Cloud providers can save considerable monitoring cost through consolidation of monitoring tasks, which is similar to the idea of server virtualization.

Nevertheless, enabling violation-likelihood based sampling is difficult. First, we must find a sound and efficient technique to *measure the violation likelihood* (VL). It should quantify VL so that VL can be linked to a certain accuracy goal. In addition, it should distinguish tasks with relatively stable value changes (e.g., the one in Figure 3(a)) and those with volatile value changes (e.g., monitored value bursts), so that users do not need to choose which task is suitable for VL based sampling.

Second, we also need *a scheme to adjust sampling intervals* based on the VL estimation. Ideally, it should make adjustment based on user-specified accuracy goal (e.g., "I can tolerate at most 1% of violation not being detected, compared with periodical sampling with 10-second intervals")[3]. Note that this gives users a new flexible and intuitive monitoring interface with automatic cost-accuracy tradeoff.

Finally, state monitoring tasks often involve multiple monitors. When a user specifies a task-level accuracy goal, how should we adjust sampling intervals on multiple monitors to satisfy the task-level accuracy? Figure 3(b) shows one task with three monitors, where each monitor observes different monitored value distributions. Should we simply set all monitors with the same sampling interval? Or is there a way that achieves the least overall sampling cost? In the rest of this section, we present our approach to address these challenges.

4.1 Measuring Violation-likelihood

Preliminaries. The specification of a monitoring task includes a default sampling interval I_d, which is also the smallest sampling interval necessary for the task. We consider the mis-detecting rate of violation is negligible when the default sampling interval is used. In addition, the specification also includes an error allowance which is an acceptable probability of mis-detecting violations. 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 can be missed. Violation likelihood at time t can be naturally defined as follows,

Definition 3: Violation Likelihood at time t is defined by $P[v(t) > T]$ where $v(t)$ is the monitored metric value at time t. In addition, we denote the violation likelihood of metric m with $\varphi(m)$.

As Figure 4 shows, the violation likelihood for the next (future) sampled value is determined by two factors, *the current sampled value* and *changes between values*. When the current sampled value is low, a violation is less likely to occur before the next sampling time. Correspondingly, a violation is more likely to occur before the next sampling if the current sampled value is high (our technique is general and does not need to distinguish increasing and decreasing changes). Similarly, when the difference between two continuously sample 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 the current sampling frequency). Let δ be the difference between the two continuously sampled

2. This alternative approach can be useful for monitoring scenarios where sampling cost is substantial and missing a small portion of violations is acceptable. Note that it may not be a good option for monitoring tasks with strict accuracy requirement.

3. We want to point out that periodical sampling can not guarantee accuracy either, because unless the sampling interval is infinitesimal, violation may happen between two consecutive sampling operations.

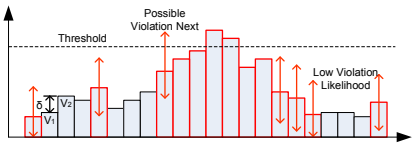


Fig. 4: Violation Likelihood Based Adaptation

values when the default sampling interval is used. We consider δ as a random variable, and values of δ are time-independent.

Since we are interested in the corresponding probability of mis-detecting violation during the gap between two consecutive sampling, we define the mis-detection rate for a given sampling interval I as follows,

Definition 4: The 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 $I(I \geqslant 1)$ is the number of default sampling intervals.

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

$$\beta(I) = 1 - \prod_{i\in[1,I]} (1 - P(v(t_1) + i\delta > T)) \tag{3}$$

We next apply Chebyshev's Inequality [18] to *efficiently* compute the upper bound of $\beta(I)$. We know that the violation likelihood for a value $v(t_2)$ that is sampled i default sampling intervals later than t_1 is $P[v(t_2) > T] = P[v(t_1) + i\delta > T]$. With Chebyshev's inequality, we have $P[\delta > \frac{T-v(t_1)}{i}] \leqslant 1/(1+(\frac{T-v(t_1)-i\mu}{i\sigma})^2)$ where μ and σ are the mean and variance of δ. Accordingly, we have

$$\beta(I) \leqslant 1 - \prod_{i\in[1,I]} \frac{(\frac{T-v(t_1)-i\mu}{i\sigma})^2}{1+(\frac{T-v(t_1)-i\mu}{i\sigma})^2} \tag{4}$$

Inequality 4 provides us the method to estimate the probability of mis-detecting violation for a given sampling interval I. We next present the dynamic sampling algorithm.

4.2 Violation-likelihood based Adaptation

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 mis-detecting rate $\beta(I)$ according to inequality 4. We denote this upper bound with $\overline{\beta(I)}$. As long as $\beta(I) \leqslant \overline{\beta(I)} \leqslant err$ where err is the specified error allowance, the mis-detecting 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 certainly to cause $\overline{\beta(I+1)} > err$. Typically, we set $\gamma = 0.2$ and $p = 20$. The sampling algorithm starts with the default sampling interval I_d, which is also the smallest possible interval. In addition, user 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. We quantitatively measure the monitoring accuracy achieved by the adaptation scheme in Section 6 and the results show that the actual mis-detection rate is consistently lower or close to the user specified error allowance.

Since computing inequality 4 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 [19]. 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 $n > 1000$.

4.3 Distributed Sampling Coordination

The monitoring algorithm we study so far adjusts the sampling interval at individual monitor level. A new problem arises when a state monitoring task involves more than one monitor, how to determine the sampling interval on individual monitors? On one hand, we must ensure that the task-level accuracy goal is achieved. On the other hand, we should also try to minimize the overall sampling cost for monitoring efficiency.

Task-Level Monitoring Accuracy. Before a coordinator reports a global state violation, it must first receive a local violation report from one of the monitors. Hence, receiving a local violation report is a necessary condition of detecting a global state violation. Let $\beta_i(I_i)$ denote the mis-detection rate of monitor i, β_c denote the mis-detection rate of the coordinator. Clearly, $\beta_c \leqslant \sum_{i\in N} \beta_i(I_i)$ where N is the set of all monitors. Therefore, as long as we limit the sum of monitor mis-detection rate to stay below the specified error allowance err, we can achieve $\beta_c \leqslant \sum_{i\in N} \beta_i(I_i) \leqslant err$.

Distributed Coordination. While there are many ways to distribute the total error allowance to individual monitors, they may not result in the same sampling cost. Take Figure 3(b) for an example, if one monitor node observes highly volatile monitored values, it may not be able to increase its sampling interval, even assigned the entire task-level error allowance (all other monitors assigned with no error allowance).

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.

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 earlier 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 $e_i = \frac{\beta(I_i)}{1-\gamma}$ (derived from the adaptation rule).

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 $\frac{err}{100}$.

In all our experiments, we find that the violation likelihood estimation consumes trivial CPU time. Hence, the cost introduced by violation likelihood estimation is negligible. The efficiency is primarily achieved by applying Chebyshev's inequality for violation likelihood estimation. In addition, estimation results of previous sampling intervals can be reused when changes of the monitoring value and the delta distribution are trivial.

5 Supporting Multi-tenancy

In this section we present a resource-aware state monitoring consolidation technique at the monitoring topology level. We use monitoring topology to refer to the overlay network that connects monitors and coordinators. Because monitors rely on the topology to deliver monitoring data, its efficiency and multi-tenancy support is critical for Cloud monitoring services. Note that we use multi-tenancy to refer to the concurrent running of monitoring tasks, rather than that of general Cloud applications. Our technique explores cost sharing opportunities across multiple monitoring tasks, and refines the state monitoring plan for each tenant's monitoring tasks through resource-sensitive monitoring topology augmentation. It not only removes duplicated monitoring workloads, but also reduces the message miss rate.

5.1 Implications of Monitoring Topology

Cloud monitoring services host a large number of state monitoring tasks for different users. Each task runs on a set of distributed nodes to monitor system or application state information with regarding to a certain attribute (e.g., response time). The set of nodes involved with different tasks often overlap with each other. In other words, a node may host multiple tasks. In most cases, tree structures are used as monitoring topology. For instance, all nodes of a task send state information to a central node where the coordinator runs (1-level tree or star-like tree).

Existing works on state monitoring over multiple nodes [20], [21] employ either static monitoring trees, where a pre-constructed tree is used for all monitoring tasks across all applications, or construct trees for individual monitoring tasks independently. While these two approaches work well on dedicated monitoring infrastructures, they have limitations for supporting multi-tenant MaaS in Cloud.

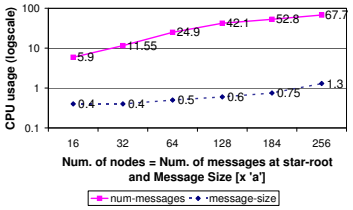


Fig. 5: CPU Usage vs Increasing Msg Num/Size

First, these two approaches are oblivious to monitoring workloads. They may construct monitoring topologies that are sub-optimal in terms of efficiency and scalability. For instance, if two tasks share the same set of nodes, using a single tree for both tasks is clearly beneficial as nodes can combine updates to reduce the number of update messages. Figure 5 shows how significant the *per-message processing overhead* is as nodes in a task increase. The measurements were performed on a machine with a 4-core 850MHz PowerPC processor. Nodes are configured in a star-like tree where each node periodically transmits a single fixed small message to a root node. 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. We also measured the CPU overhead as the message size increases as shown in the lower curve. The cost incurred to receive a single small message of size a is around 0.2% while the cost incurred to receive a single message of size $256a$ is still around 1.4% (see Figure 5). Therefore, reducing the number of monitoring messages by combining monitoring value updates into fewer number of messages is important for efficiency due to the significance of per-message overhead.

Second, neither approach considers the per-node available resources or the monitoring workload. As a result, when some nodes experience heavy workloads, monitoring tasks may suffer from unexpected message losses, which leads to undesired monitoring errors. This problem may get aggravated for large scale monitoring tasks with many nodes, or requiring collecting and sending detailed state data to the coordinator at high frequencies. Therefore, to support multi-tenancy in MaaS, monitoring topology construction must exploit cost-sharing opportunities for efficiency and scalability, and at the same time, matches node-level monitoring workload with available monitoring capacities.

5.2 Monitoring Topology Planning

Finding the optimal monitoring topology for a given set of tasks is difficult. In fact, it can be shown that this problem is NP-complete [22]. We next illustrate the problem with an example in Figure 6 where we have 6 monitoring nodes in the system and each monitors a set of attributes indicated by alphabets on nodes.

Example (A), (B) and (C) show three widely used topologies. (A) shows the star topology where every nodes send their updates directly to the central node where coordinators of different tasks are hosted. It has poor scalability, because the central node consumes considerable resources for per-message overhead. (B) shows a single tree topology where all nodes are included. While it reduces the per-message processing overhead

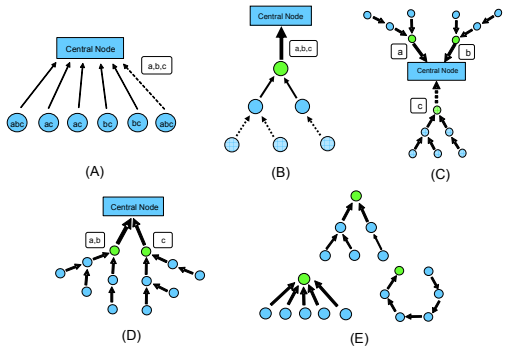


Fig. 6: Motivating examples of topology planning

on both the central node and the root node of the tree, nodes in the tree have to relay monitoring data for *all* tasks. Hence, the root node easily gets overloaded by relaying cost. (C) shows a per-task tree topology where a separate tree is built to organize monitor nodes of a single task. It addresses the issue in (A) and (B). However, if a monitor node runs multiple tasks, it has to join multiple trees and send update messages separately for different tasks, instead of combining the updates. This is clearly inefficient due to per-message overhead. As a result, nodes running multiple tasks may soon become bottleneck given increasing tasks.

One possible approach is to partition tasks into clusters where tasks with similar set of nodes form a tree. In example (D), nodes are partitioned into two trees. One tree for task a and b, another for task c. This gives service providers the flexibility to achieve maximum monitoring efficiency by combining updates of different tasks. Furthermore, Within each cluster, we can construct trees based on the available resource on each node (example (E)). As Cloud nodes dedicate most resources to hosting applications, it is crucial that no nodes gets overloaded with monitoring workload, which in turn ensures performance isolation of monitoring services. We below present details of this approach.

5.3 Resource-Aware Topology Planning

Our resource-aware topology planning approach adopts *a guided local search* method. It starts with an initial topology where a separate tree is created for each task (example (C)), and iteratively merges (or splits) trees to optimize the monitoring topology based on two prioritized goals until no improvement can be done. The primary goal is to maximize the number of tasks supported by the topology, which emphasizes service scalability. The secondary goal is to minimize the overall data delivery cost (both per-message and relay cost), which tries to save resources for future tasks. The local search based approach is the key for addressing the complexity of the problem. Furthermore, it naturally fits Cloud monitoring services, because it can continuously improve the monitoring topology as monitoring tasks are added or removed based on the current topology without starting from scratch.

For each iteration, our approach selects the most promising merging or splitting operation by estimating the resulting performance gain of the topology. A straightforward alternative is to try all possible merging and splitting operations and pick the best one. However, this method is not practical due to high complexity in candidate space and tree construction. Once an operation is selected, it uses an adaptive tree construction algorithm to build a resource-aware tree where every node has sufficient capacity to support all its tasks. Note that the iterative planning process is virtual in the sense that it only constructs the overlay in memory and estimates the cost and performance of the resulting overlay. Monitors are instructed to construct the planned overlay only after the planning process finishes.

Guided Local Search. The overall monitoring data delivery cost consists of two parts, the per-message processing overhead and the relay cost. Hence, the estimation of efficiency gain for a merging or splitting operation should consider the change in both the per-message processing overhead and the relay cost. Let $g(m)$ be the overall reduction in cost of a merging/splitting operation m, $\Delta c_p(m)$ be the estimated difference in per-messaging processing cost due to m and $\Delta c_r(m)$ be the estimated difference in relay cost due to m. We have $g(m) = \Delta c_p(m) + \Delta c_r(m)$. To perform efficient estimation, we assume an aforementioned star topology can accommodate all nodes in the resulting tree. We use A_i to denote the tree associated with attribute set i. Let C be the per-message overhead, a be the cost of a message of unit size, and N_{A_i} denote the set of nodes associated with tree A_i.

The following equations capture the estimated difference in both overhead cost ($\Delta c_p(m)$) and relay cost ($\Delta c_r(m)$) due to m respectively:

$$\Delta c_p(m) = \begin{cases} (-1) \cdot C \cdot |N_{A_i} \cap N_{A_j}| & A_i + A_j = A_k \\ C \cdot |N_{A_j} \cap N_{A_k}| & A_i - A_j = A_k \end{cases}$$

$$\Delta c_r(m) = \begin{cases} a \cdot |N_{A_i \cup A_j} - N_{A_i \cap A_j}| & A_i + A_j = A_k \\ (-1) \cdot a \cdot |N_{A_i} - N_{A_i \cap A_j}| & A_i - A_j = A_k \end{cases}$$

Here $A_i + A_j$ denotes the operation of merging tree A_i and A_j. Correspondingly, $A_i - A_j$ means the operation of removing attribute set j from the attribute set i in tree A_i. A_k represents the resulting tree of merging or splitting operations. Intuitively, when we merge two trees, the per-message processing cost reduces as nodes associated with both trees 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 a tree, the per-message overhead cost increases and the relaying cost decreases. The above equations capture these two changes and make the estimation possible. Our current implementation uses monitoring traffic volume to specify per-node monitoring capability. However, our approach can support other types of resources.

Adaptive Tree Construction. We employ an adaptive tree construction algorithm that iteratively invokes two procedures, the construction procedure and the adjusting procedure. In the construction procedure, we use a scheme that builds star-like trees. The star scheme gives the priority to increasing the breadth of the tree. The star scheme creates bushy trees and pays low relay cost. However, the node with large degree, suffers from high per-message overhead. Hence, when such nodes are overloaded, we invoke the adjusting procedure. Intuitively, the adjusting procedure "streches" the tree by decreasing the breadth of the tree and increasing the height of the tree, which helps achieve good per-message load balance, with the cost of slightly increased relay cost. Due to the space constraint, we refer readers to [22] for details.

The topology planning technique primarily focuses on fair monitoring resource usage across distributed machines. It is an alternative approach to meet diverse monitoring needs in MaaS, and is particularly useful for data-intensive environments where monitoring resource consumption is non-trivial. Other topologies such as static ones may be more desirable for

scenarios where monitoring resource consumption is not the concern and simple, fixed hierarchy is preferred.

6 EVALUATION

We highlight the most important observations in our experiments as follows. First, our window-based violation detection algorithm saves 50% to 90% communication cost compared with previous detection algorithms and simple alternative schemes. Its parameter tuning ability is the key to achieve low communication cost for various monitoring environments. Second, the violation-likelihood based sampling technique reduces considerable monitoring cost and the actual monitoring accuracy loss is smaller or very close to user specified error allowance. Finally, our monitoring topology planning technique improves the scalability of the monitoring service by minimizing monitoring data delivery overhead.

6.1 Experiment Setup

Our current implementation of the MaaS prototype is written with Java 1.6. It includes monitoring daemons, monitor servers, coordinator servers and a simple management center. As the name suggests, a monitor server hosts monitors from different tasks and a coordinator server hosts coordinators from different tasks. Monitor and coordinator servers can run on production servers to provide monitoring services. The monitor server uses a thread pool to efficiently support a large number of monitors. Monitors are sorted based on their next scheduled sampling time. A monitor is scheduled to run with a thread only when it reaches its sampling execution time. We leverage an open source job scheduling library, Quartz 1.8.3 [23], to implement the thread pool. We implement the detection algorithm and the VL based sampling technique at the monitor and the coordinator level, and the topology planning algorithm on the coordinator server.

We deploy the prototype on Emulab [24]. Specifically, we install a large number of monitoring daemons on 30 relatively low end machines (Intel Pentium III 600 MHz CPU and 256MB memory). These monitoring daemons together emulate a Cloud infrastructure for the monitoring system as daemons mimic physical/virtual machines, routers and applications by reporting the corresponding monitoring information when requested. Daemons are fed with the trace data (described later) collected at system, network and application levels in production environments. On the monitoring system side, we deployed 10 monitor servers and 1 coordinator server on 11 relatively high end Emulab machines (Intel Core 2 Duo 2.4GHz and 2048 MB memory).

We utilize both real world system, network and application traces and synthetic traces to evaluate our techniques. The real world traces are collected from production environments to evaluate the effectiveness of our approach for the monitoring of real world systems and networks. For the system level monitoring, we use a large scale performance trace [25] collected from 300 Planetlab [26] nodes over several months. This trace contains performance data on 66 system attributes. For the network level monitoring, we use a series of netflow traces collected from core routers in Internet2 [27]. A flow in the trace records the source and destination IP addresses (sanitized) as well as the traffic information (total bytes, number of packets, protocol, etc.) for a flow of observed packets. The trace data contain approximately 42,278,745 packet flows collected from 9 core routers in the Internet2 network. For the application level traces, we use *WorldCup* which contains real world traces of HTTP requests across a set of distributed web servers. The trace data come from the organizers of the 1998 FIFA Soccer World Cup [14] 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.

We use synthetic monitoring traces to create various monitoring conditions which are difficult to obtain with real world collected traces. 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.

6.2 Results

6.2.1 Distributed Violation Detection

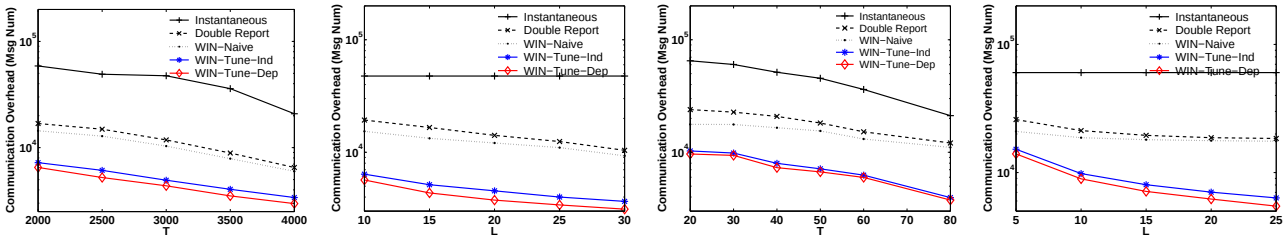
Comparison of communication efficiency. Figure 7 compares the communication overhead of our window based detection algorithm (WIN-Tune) with that of the instantaneous detection algorithm (Instantaneous), a simple alternative window based detection scheme (Double Report) where monitors report both the beginning and the ending of a local violation period, and window based detection with naive parameter setting (WIN-Naive) for the World Cup dataset and Synthetic dataset. In addition, (Ind) and (Dep) denote the time independent and time dependent model used in parameter tuning. The first model is simple but less accurate, while the second one is accurate but more expensive to use. We vary T and L in a way that the total length of global violations 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 7(a) shows the total message number generated by WIN-Tune is nearly a magnitude lower than that of the instantaneous approach. Double Report and WIN-Naive, while outperform the instantaneous approach as they delay global polls, generate more traffic compared with WIN combined with our parameter tuning scheme (WIN-Tune). Double Report suffers from frequent reporting for short violation periods, especially when T is small. WIN-Naive fails to achieve better efficiency because it does not explore different value change patterns at different nodes. 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 7(b), while the instantaneous approach is not benefited from large values of L, the WIN algorithm pays less communication overhead as L grows, since nodes increase filtering window sizes and the coordinator rules out more global polls with increasing L. Figure 7(c) and 7(d) show similar results for the Synthetic dataset.

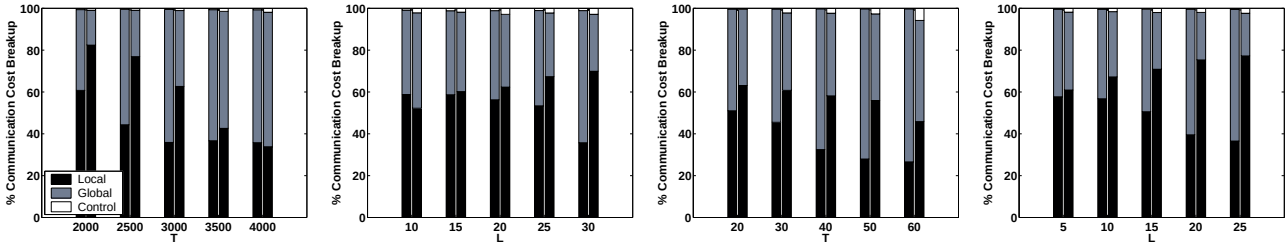
Communication cost breakup analysis. Figure 8(a) and 8(b) show communication cost breakup of WIN for the WorldCup dataset, where communication overhead is divided into three parts: local violation reporting, global polls, and control. Control messages are mainly generated by the parameter tuning scheme. 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 8(a), as T grows, the portion of global poll communication steadily increases, as local violations occur less frequently. The portion of control communication also increases,



(a) WorldCup - Increasing T (b) WorldCup - Increasing L (c) Synthetic - Increasing T (d) Synthetic - Increasing L

Fig. 7: Comparison of Communication Efficiency in Terms of Message Number



(a) WorldCup - Increasing T (b) WorldCup - Increasing L (c) Synthetic - Increasing T (d) Synthetic - Increasing L

Fig. 8: Communication Cost Breakup (Message Volume and Message Number)

due to the reduction of local violation reporting and global polls. Similarly, in Figure 8(b), we observe the growth of the global poll portion along with increasing L, as nodes increase p_i to filter more reports. Figure 8(c) and 8(d) provide similar results for the Synthetic dataset.

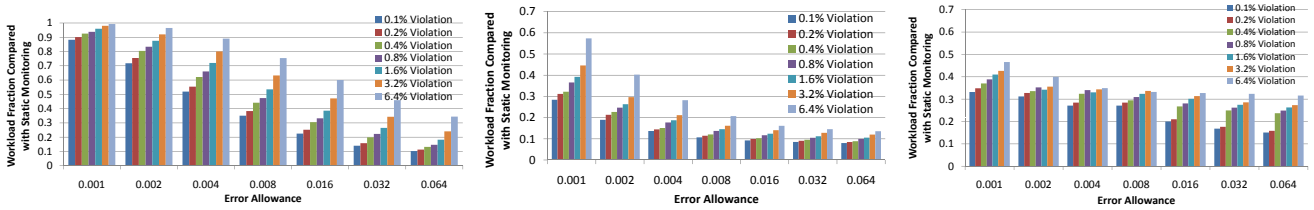
6.2.2 *Local State Monitoring*

Efficiency. The first set of experiments evaluate the efficiency of our violation likelihood based sampling techniques. Figure 9(a) shows the results for system level monitoring, where we feed the Planetlab monitoring traces into daemons running on a set of hosts in Emulab in order to emulate the 300-node monitoring environment. We are interested in the ratio of sampling operations (Y axis) between VL based sampling and periodical monitoring with fixed sampling intervals. We vary both the error allowance (X axis) and the alert state selectivity k (percentage of violations) in monitoring tasks (denoted by different series) to test their impacts to monitoring efficiency. Here the error allowance is the allowed percentage of the ratio of miss-detected state alerts. We can see that VL techniques effectively reduces monitoring overhead by 10%-90%. Clearly, the larger the error allowance, the more sampling operations VL can save by reducing monitoring frequency as long as the probability of mis-detection is lower than the error allowance. The alert state selectivity k also plays an important role, e.g., varying k from 6.4% to 0.1% leads to almost 40% cost reduction. This is because lower k leads to fewer state alerts, which allows VL to use longer monitoring intervals when previous observed values are far away from the threshold (low violation likelihood). Note that for many real world state monitoring tasks, the selectivity k is often quite small. For instance, for a monitoring task with a default 10-second monitoring interval, generating one alert event per hour leads to a $k = 1/360 \approx 0.003$. Hence, we expect VL to save considerable overhead for most monitoring tasks.

Figure 9(b) illustrates the results for our network monitoring experiments. We perform a heavy hitter detection [5] monitoring to emulate a real network monitoring scenario. Specifically, we distribute the packet flows captured in the trace data to monitoring daemons based on destination addresses. Among all packet flows, we select those with the top 10% number of packet to create state monitoring tasks. Similar to the monitoring tasks used in the system level monitoring experiment, these tasks report state alerts when the packet number of the corresponding flow exceeds a threshold (determined by the selectivity k). The results suggest that VL also effectively reduces the monitoring overhead for Heavy Hitter detection, with even higher cost saving ratios compared with those in the system monitoring experiments. This is because the changes of traffic are often less volatile compared the changes of system metric values (e.g., CPU utilization). This is especially true for network traffic observed at night. As a result, our temporal adaptation techniques can employ relatively long monitoring interval.

We show the results of application level monitoring in Figure 9(c). This experiment emulates an application level workload monitoring where monitoring daemons are fed with http access traces and monitoring tasks aim at detecting the burst of access. Specifically, we create two types of application-level monitoring tasks for distributed admission and resource provisioning. Distributed admission tasks monitors whether the workload, measured by request rates and the associated network IO, generated by a set of users exceeds a given threshold. Resource provisioning tasks watch the workload on different objects requested by users and raise alerts when the workload on a certain object exceeds a threshold. We use these two types of monitoring tasks as examples of application level monitoring tasks. We observe similar cost saving results in this figure. The high cost reduction achieved in the application level monitoring is due to the burst nature of accesses (high access rate during games, and low access rate during other time). It allows our adaptation to use large monitoring interval during off-peak time. We speculate that our techniques can benefit from this type of change patterns in many other applications (e.g., e-business websites) where diurnal effects and access bursts are common.

Monitoring Accuracy. We also quantitatively measure the accuracy achieved by our technique. Figure 10 shows the actual mis-detection rate of alerts for the system-level monitoring experiments. We can 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 rate. There are two



(a) System Level Monitoring (b) Network Level Monitoring (c) Application Level Monitoring

Fig. 9: Monitoring Overhead Saving under Different Error Allowance and State Alert Rates

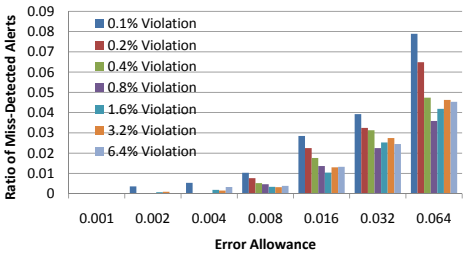


Fig. 10: The Actual Mis-Detection

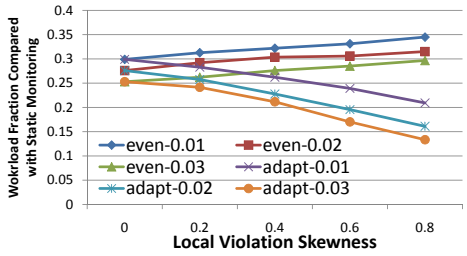
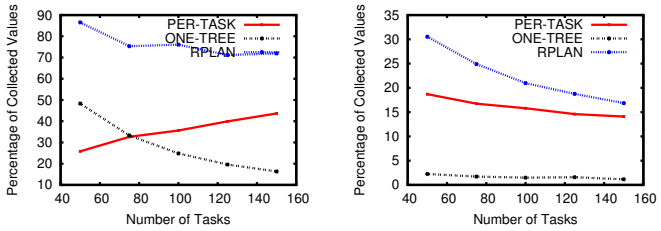


Fig. 11: Distributed Sampling Coordination



(a) Increasing Small-scale Tasks (b) Increasing Large-scale Tasks

Fig. 12: Tree Partitioning under Diff. Workload

reasons for this. First, high alert selectivity leads to few overall alerts which drives up the mis-detection rate even for fixed number of mis-detections. Second, high alert selectivity also makes VL prefer low monitoring frequency which increases the chance of missing alerts. We do not show the corresponding results for network and application level monitoring due to the space limit, as the results are similar.

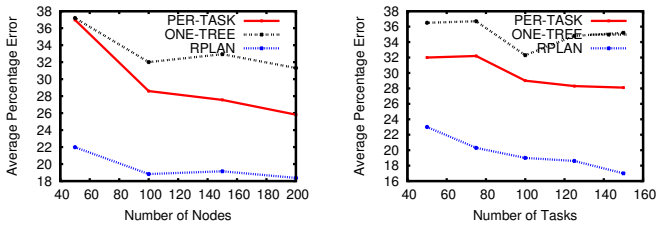
Distributed Sampling Coordination. Figure 11 illustrates the detailed performance of different error allowance distribution schemes over the netflow trace. To vary the cost reduction yield on monitors, we change the local violation rates by varying the local thresholds on different nodes. Initially, we set each monitor a local threshold so that all monitors have the same local violation rate. We then gradually change the local violation rate distribution to a Zipf distribution [28] (the x-axis shows the skewness of the distribution and the distribution is uniform when skewness=0). Zipf distribution is commonly used to simulate skewed distribution in real world. We compare the performance of our iterative tuning scheme (adapt) described with an alternative scheme (even) which always divides the global error allowance evenly among monitors.

Both schemes have similar cost reduction performance when monitors have the same local violation rates, because the cost reduction yields are the same across monitors. However, the cost reduction of even scheme gradually degrades with increasing skewness of the local violation rate distribution, 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 achieves considerable better yield as it continuously allocates error allowance to monitors with high yield.

6.2.3 *Multi-tenancy Support*

Varying the scale and number of monitoring tasks. Figure 12 compares the performance of different topology construction approaches under different workload characteristics. This set of experiments utilizes synthetic data. We generate monitoring tasks by randomly selecting monitored attributes and a subset of nodes with uniform distribution. We perform stress test and measure the percentage of delivered data to evaluate the efficiency of topology. In Figure 12(a), where we increase the number of small scale tasks, our resource-aware planning (RPLAN) performs consistently better than per-task tree (PER-TASK) and single tree (ONE-TREE) schemes. In addition, ONE-TREE outperforms PER-TASK when task number is small. As each node sends only one message which includes all its own data and those received from its children, ONE-TREE causes the minimum per-message overhead. However, when tasks increase, the capacity demand of low level nodes, i.e., nodes close to the root, increases significantly, which limits the size of the tree and causes poor performance. In Figure 12(b), where we increase large-scale tasks to create heavy workloads, RPLAN gradually converges to PER-TASK, as PER-TASK provides the best load balance which results in the best performance in this case.

To evaluate the real world performance of our topology planning techniques, we measure the average percentage error of received attribute values for synthetically generated tasks. Specifically, we obtain the true monitoring values at the end of the experiment by combining local log files. The average percentage error is measured by the difference between the true monitoring value and the corresponding value observed by our scheme. This experiment consisted of over 200 monitors, with 30-50 attributes to be monitored on each node. Figures 13(a) compares the achieved percentage error between different partition schemes given increasing nodes. The figure shows that our partition augmentation scheme outperforms the other partition schemes. The percentage error achieved by RPLAN is around 30%-50% less than that achieved by PER-TASK and ONE-TREE. We observe similar results in Figure 13(b) where we increase the number of running tasks.



(a) Increasing Nodes (b) Increasing Tasks

Fig. 13: Comparison on Average Percentage Error

6.2.4 *Putting things together*

The last set of experiments evaluate all three techniques together in a distributed rate limiting scenario (described in Section 2.2) where we distribute the aforementioned HTTP access log to a set of monitoring nodes to simulate a Cloud application running across distributed nodes. We tag each access with a user ID (drawn from a group of 100 IDs following the Zipf distribution) so that the global access rate of a user can be tracked by monitoring the corresponding access on each node. We employ the window-based state monitoring model with the threshold set to the 90th percentile of monitored values and the monitoring window set to 15. In addition, we use a standalone coordinator to collect and process monitoring messages for all monitoring tasks. Figure 14 shows the combined performance improvement contributed by our three techniques given increasing nodes, where the y axis shows the percentage of saved communication volume (over instantaneous state monitoring), the percentage of saved sampling operations (over periodical sampling) and the percentage of nodes which exceed their local monitoring bandwidth limit for the window based violation detection scheme, the violation likelihood based sampling scheme and topology planning schemes respectively.

The window-based state monitoring significantly reduces the inter-node communication volume (65%-87%). Note that such savings are especially important when nodes are distributed in geographical datacenters. Furthermore, the saving generally grows with increasing number of nodes as the lazy global poll often avoids expensive global collection of all local monitoring values. The violation likelihood based sampling also substantially lowers the total number of sampling performed on each node (about 60%). The tasks are set to tolerate no more than 1% mis-detected (instantaneous) violations. As window-based state monitoring captures continuous violations, the actual mis-detection rate is well below 1%. In addition, the scheme is benefited from the separated periods of high/low access rates (which is quite common for many applications) as the scheme can maintain low sampling rates for a relatively long time. The resource-aware topology planning technique safeguards the per-node monitoring bandwidth usage with increasing nodes, while the Per-Task scheme introduces high per-message overheads and the One-Three scheme causes heavy local bandwidth usage on nodes close to the root. Note that as the number of messages emitted by each node is dynamic, we use the upper bound for per-node resource usage estimation. This ensures low percentages of per-node over-consumption, but also makes the scheme to favor "tall" trees (more relay hops). For monitoring scenarios that are highly sensitive to the Time-To-Observe latency, one can use less conservative estimation (e.g., using average communication cost).

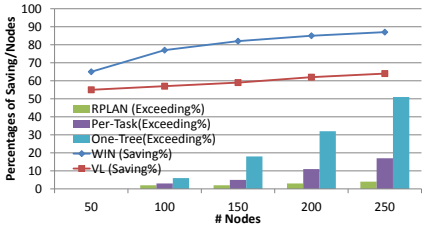


Fig. 14: Combined Monitoring Performance

7 Related Work

Existing works [11], [12], [13] on global violation detection focus on communication efficiency algorithms based on an instantaneous detection model. We show that MaaS should also provide window based detection as an important functionality. Compared with the instantaneous model, window based detection model not only gains advantages in avoiding false positives introduced by outliers and noises in monitoring data, but also achieves even better communication efficiency.

Compared with global violation detection, local state monitoring is a problem receives less attention. Existing works on global violation detection [11], [12], [13] as well as distributed stream monitoring works [29], [30] assume that the cost of local state monitoring is trivial. However, this assumption may not hold for monitoring tasks such as network traffic monitoring where local state monitoring cost is non-negligible. As MaaS must support monitoring tasks with heavily diverse local state monitoring costs, our violation-likelihood based monitoring technique provides a flexible framework that enables accuracy controlled dynamic local state sampling.

A number of existing works in sensor network area study using correlation to minimize local state monitoring cost in terms of energy consumption on sensor nodes [31]. Our approach distinguishes itself from these works in several aspects. First, these works often reply on the broadcast feature of sensor networks, while Cloud datacenter networks are very different from sensor networks where broadcast is often not an option. 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, our MaaS must support a wide rang of tasks as a multi-tenant platform. Finally, some works [32] make assumption on value distributions, while our approach makes no such assumptions.

Existing works on processing multiple monitoring tasks often either focus on eliminating duplicated workload on a centralized monitoring server [33], [34], [35], [36], or study efficient algorithms for continuous aggregation queries over physically distributed data streams [37], [38], [39], [40], [41]. However, these works either rely on centralized servers or assume the monitoring topology are provided as part of the input. Furthermore, existing works [31], [42] do not consider monitoring resource consumption at each node which may cause inter-monitoring-task interference or monitoring-application interference in a multi-tenant Cloud monitoring environment. Our resource-aware planning algorithm solves the joint problem of resource-constrained monitoring and multi-task optimization, and achieves significant benefit over solving only one of these problems in isolation.

8 Conclusions and Future Work

Monitoring-as-a-service (MaaS) is a critical functionality for effective Cloud management. We argue that MaaS should support not only the conventional state monitoring capabilities

commonly used in Cloud datacenters today, but also provide performance-enhanced functionalities that allow reducing monitoring cost, improving monitoring scalability, and strengthening the effectiveness of monitoring service consolidation and isolation. In this paper we present three enhanced MaaS capabilities: (i) window based violation detection as an alternative enhancement of instantaneous violation detection, (ii) violation likelihood based state monitoring as an alternative enhancement of period state monitoring, and (iii) multi-tenant state monitoring techniques as an alternative enhancement of single tenant approach. Through experimental evaluation we show that the three enhanced MaaS capabilities offer complimentary and yet effective state monitoring capabilities with high efficiency, scalability and multi-tenancy support. We argue that MaaS for Cloud management needs to provide both conventional and enhanced state monitoring capabilities to support different monitoring service requirements.

We would like to conclude by stating that this work presents only one step towards a truly scalable and customizable MaaS. Many issues need to be investigated in depth for MaaS to be a successful service computing metaphor for Cloud state management, such as robustness, fault-tolerance, security, privacy as well as elasticity to meet different and possibly dynamically changing needs of state monitoring and state-based resource management from a variety of Cloud applications and Cloud service providers.

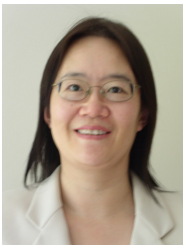
ACKNOWLEDGMENTS

This work is partially supported by NSF grants from CISE CyberTrust program, NetSE program and CyberTrust Cross Cutting program, and an IBM faculty award and an Intel ISTC grant on Cloud Computing.

REFERENCES

[1] B. Raghavan, K. V. Vishwanath, S. Ramabhadran, K. Yocum, and A. C. Snoeren, "Cloud control with distributed rate limiting," in *SIGCOMM*, 2007, pp. 337–348.
[2] L. Raschid, H.-F. Wen, A. Gal, and V. Zadorozhny, "Monitoring the performance of wide area applications using latency profiles," in *WWW'03*.
[3] G. Gu, R. Perdisci, J. Zhang, and W. Lee, "Botminer: Clustering analysis of network traffic for protocol- and structure-independent botnet detection," in *USENIX Security Symposium*, 2008, pp. 139–154.
[4] G. Ren, E. Tune, T. Moseley, Y. Shi, S. Rus, and R. Hundt, "Google-wide profiling: A continuous profiling infrastructure for data centers," *IEEE Micro*, vol. 30, no. 4, pp. 65–79, 2010.
[5] N. Jain, P. Mahajan, D. Kit, P. Yalagandula, M. Dahlin, and Y. Zhang, "Network imprecision: A new consistency metric for scalable monitoring," in *OSDI*, 2008, pp. 87–102.
[6] Amazon, CloudWatch, http:// aws.amazon.com/ cloudwatch/.
[7] C. Estan and G. Varghese, "New directions in traffic measurement and accounting," in *SIGCOMM*, 2002, pp. 323–336.
[8] Amazon, "Ec2 auto-scaling," http://aws.amazon.com/autoscaling/.
[9] Wikipedia, "Cloudkick," http://en.wikipedia.org/wiki/Cloudkick, 2012.
[10] Cloudyn, http://cloudyn.com/how-it-works/under-the-hood, 2012.
[11] M. Dilman and D. Raz, "Efficient reactive monitoring," in *INFOCOM*, 2001, pp. 1012–1019.
[12] R. Keralapura, G. Cormode, and J. Ramamirtham, "Communication-efficient distributed monitoring of threshold counts," in *SIGMOD*, 2006.
[13] S. Agrawal, S. Deb, K. V. M. Naidu, and R. Rastogi, "Efficient detection of distributed constraint violations," in *ICDE*, 2007, pp. 1320–1324.
[14] "1998 worldcup website logs," http://www.acm.org/sigcomm/ITA/.
[15] "Ntp," http://www.eecis.udel.edu/ mills/ntp/html/index.html.
[16] S. Meng, T. Wang, and L. Liu, "Monitoring continuous state violation in datacenters: Exploring the time dimension," in *ICDE*, 2010, pp. 968–979.
[17] S. Meng, A. Iyengar, I. Rouvellou, L. Liu, K. Lee, B. Palanisamy, and Y. Tang, "Reliable state monitoring in cloud datacenters," in *IEEE Cloud*, 2012.
[18] G. Grimmett and D. Stirzaker, *Probability and Random Processes 3rd ed.* Oxford, 2001.
[19] D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms, 3rd Ed.* Addison-Wesley, 1998.
[20] S. Madden and et al., "Tag: A tiny aggregation service for ad-hoc sensor networks," in *OSDI*, 2002, pp. 131–146.
[21] P. Yalagandula and M. Dahlin, "A scalable distributed information management system," in *SIGCOMM04*, pp. 379–390.
[22] S. Meng, S. R. Kashyap, C. Venkatramani, and L. Liu, "Remo: Resource-aware application state monitoring for large-scale distributed systems," in *ICDCS*, 2009, pp. 248–255.
[23] "Quartz," http://www.quartz-scheduler.org/.
[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," Boston, MA, Dec. 2002.
[25] Y. Zhao, Y. Tan, Z. Gong, X. Gu, and M. Wamboldt, "Self-correlating predictive information tracking for large-scale production systems," in *ICAC*, 2009, pp. 33–42.
[26] L. L. Peterson, "Planetlab: Evolution vs. intelligent design in planetary-scale infrastructure," in *USENIX Annual Technical Conference*, 2006.
[27] Internet2, http://www.internet2.edu/observatory/archive.
[28] L. A. Adamic and B. A. Huberman, "Zipfs law and the internet," *Glottometrics*, vol. 3, pp. 143–150, 2002.
[29] C. Olston, J. Jiang, and J. Widom, "Adaptive filters for continuous queries over distributed data streams," in *SIGMOD*, 2003, pp. 563–574.
[30] C. Olston and J. Widom, "Offering a precision-performance tradeoff for aggregation queries over replicated data," in *VLDB*, 2000, pp. 144–155.
[31] A. Silberstein, R. Braynard, and J. Y. 0001, "Constraint chaining: on energy-efficient continuous monitoring in sensor networks," in *SIGMOD Conference*, 2006, pp. 157–168.
[32] D. Chu, A. Deshpande, J. M. Hellerstein, and W. Hong, "Approximate data collection in sensor networks using probabilistic models," in *ICDE*, 2006, p. 48.
[33] R. Z. 0003, N. Koudas, B. C. Ooi, and D. Srivastava, "Multiple aggregations over data streams," in *SIGMOD*, 2005, pp. 299–310.
[34] S. Krishnamurthy, C. Wu, and M. J. Franklin, "On-the-fly sharing for streamed aggregation," in *SIGMOD*, 2006, pp. 623–634.
[35] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: efficient evaluation of sliding-window aggregates over data streams," *SIGMOD Record*, vol. 34, no. 1, pp. 39–44, 2005.
[36] S. Madden and et al., "Continuously adaptive continuous queries over streams," in *SIGMOD*, 2002, pp. 49–60.
[37] R. Huebsch and et al., "Sharing aggregate computation for distributed queries," in *SIGMOD*, 2007, pp. 485–496.
[38] R. Gupta and K. Ramamritham, "Optimized query planning of continuous aggregation queries in dynamic data dissemination networks," in *WWW*, 2007, pp. 321–330.
[39] N. Trigoni and et al., "Multi-query optimization for sensor networks," in *DCOSS*, 2005, pp. 307–321.
[40] A. Silberstein and J. Yang, "Many-to-many aggregation for sensor networks," in *ICDE*, 2007, pp. 986–995.
[41] S. Xiang, H.-B. Lim, K.-L. Tan, and Y. Zhou, "Two-tier multiple query optimization for sensor networks," in *ICDCS*, 2007, p. 39.
[42] R. H. et al., "The architecture of pier: an internet-scale query processor," in *CIDR*, 2005, pp. 28–43.

Shicong Meng is currently a PhD student in the College of Computing at Georgia Tech. He is affiliated with the Center for Experimental Research in Computer Systems (CERCS) where he works with Professor Ling Liu. His research focuses on performance, scalability and security issues in large-scale distributed systems such as Cloud datacenters. He has recently been working on several projects on Cloud datacenter monitoring and management, with a strong emphasis on Cloud monitoring services.



Ling Liu is a full Professor in the School of Computer Science at Georgia Institute of Technology. There she directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of data intensive systems with the focus on performance, availability, security, privacy, and energy efficiency. Prof. Liu has published over 300 International journal and conference articles in the areas of databases, distributed systems, and Internet Computing. She is a recipient of the best paper award of ICDCS 2003, WWW 2004, the 2005 Pat Goldberg Memorial Best Paper Award, and 2008 Int. conf. on Software Engineering and Data Engineering. Prof. Liu has served as general chair and PC chairs of numerous IEEE and ACM conferences in data engineering, distributed computing, service computing and Cloud computing fields and is a co-editor-in-chief of the 5 volume Encyclopedia of Database Systems (Springer). She is currently on the editorial board of several international journals. Dr. Liu's current research is primarily sponsored by NSF, IBM, and Intel.