Polling Schedule Optimization for Adaptive Monitoring to Scalable Enterprise Systems

Fumio Machida, Masahiro Kawato, Yoshiharu Maeno
NEC Service Platforms Research Laboratories
1753, Shimanumabe, Nkahara-ku, Kawasaki, Knagawa 211-8666, Japan
{h-machida@ab, m-kawato@ap, y-maeno@aj}.jp.nec.com

Abstract

Adaptive monitoring is a promising technique to automate configurations of a monitoring server in enterprise systems according to the dynamic system reconfigurations such as server scale-out and virtual machine migration. Even after the system reconfiguration, the monitoring server need to be configured properly for providing the fresh information to clients with stabilized server load. In this paper, we propose an adaptive monitoring system that automatically changes the monitoring schedule to satisfy the required freshness under the limited server load after system reconfigurations. The adaptive monitoring system consists of a polling-based monitoring architecture and an algorithm for polling schedule generation. Since the problem for polling schedule generation is classified in NP-hard, we propose an approximation algorithm. According to the results from the experiments with real system reconfiguration scenarios, the adaptive monitoring system improves the variation coefficients of changes of CPU usages and network traffics in the monitoring server by at most 80%. We extend the proposed adaptive monitoring system to be scalable by introducing a hierarchical architecture.

Keywords: Adaptive monitoring, Polling schedule, Virtualization, System reconfigurations, Information freshness

1. Introduction

The emergence of virtual machine technologies enlarges the flexibility of the current enterprise systems. Virtual machine software such as Xen [8], VMware Infrastructure [22] and Microsoft Virtual Center [23] offer a function to create multiple execution environments on a single computer. Enterprise systems can be scale out easily by using virtual machine software and creating a virtual machine on the existing physical environments. System reconfigurations like change of server allocation, server scale out, components replacement and software updates are usually required in common enterprise system administration. Virtual machine can reduce the troubles related to hardware during system reconfigurations because virtual machine does not depend on the physical devices directly.

Although virtual machine enables easy system reconfigurations, frequent system reconfigurations increase administrative operations for the management systems to adapt to the reconfigured target systems. For example, when an administrator adds some virtual machines to the existing systems, he or she has to register the additional targets to monitoring systems or some management tools, and apply appropriate settings. The process of the reconfiguration can be executed automatically by using virtual machines. However, registrations and configuration changes of existing systems need manual operations of administrators. Configuration changes after system reconfigurations are especially important for monitoring systems. Missing registrations and improper setting of monitoring intervals lead to the degradation of the availability and performance of the systems.

We proposed an adaptive monitoring system to reduce administrative operations for reconfigurable enterprise systems. The reduction of the operations for the monitoring settings after system reconfigurations enables easy and speedy adaptation to the target systems. The proposed method generates a monitoring schedule that is a set of monitoring setting satisfying the required freshness of the monitored information and the limited monitoring server load. The system administrator does not need to estimate the impact on the performance and the availability result from the

change of monitoring settings. The schedule generation problem is an integer programming that is classified as NP-hard [7]. If the target system consists of dozens of servers, an optimal schedule is not computable in realistic time. Therefore, we proposed an approximation algorithm for schedule generation. The proposed algorithm generates an optimal schedule under a specific condition. Furthermore, we extend the proposed adaptive monitoring system to be scalable by introducing a hierarchical architecture. A single monitoring server is not realistic for managing thousands of monitoring targets in terms of the load of monitoring server. In the monitoring system using multiple monitoring servers, the query turnaround time and information freshness depend on the number of transit monitoring servers and schedules. To satisfy the requirements from clients for query response time and information freshness, the schedules for multiple monitoring servers need to be optimized. We formulized the problem to decide schedules for multiple monitoring servers configured hierarchically and an approximation algorithm to solve the problem.

The rest of this paper is organized as follows. Section 2 describes the requirements for an adaptive mechanism for monitoring server in enterprise systems. Section 3 presents our adaptive monitoring architecture and an algorithm for polling optimization. Section 4 shows experimental results. Section 5 describes the extension of the adaptive monitoring system and the schedule generation algorithm. Section 6 describes related work and, finally, Section 7 provides the conclusion.

2. Enterprise System Monitoring

Most of enterprise systems have monitoring systems to manage system resources such as servers, network devices, storages and applications. Some commercial products such as HP OpenView Network Node Manager (NNM) [17] and IBM Tivoli NetView [18] provides functions for monitoring resources based on (Simple Network Management Protocol) SNMP [10]. ZABBIX[19], OpenNMS [20] and Nagios [21] come to be known as powerful free monitoring tools that can be used for enterprise-level systems.

Adaptive monitoring appeared in our previous work is a promising technique for enterprise systems to adapt to the change of system configurations and states [1]. The number of monitoring targets in enterprise systems increases and changes dynamically according to the system reconfiguration caused by business requirements and system upgrades. The adaptive monitoring system reduces the administrative operations for monitoring server by automatically optimizes the monitoring configurations at the system reconfigurations. Since virtual machines allow easy system reconfiguration, the concept of adaptive monitoring is especially important in the consolidated server environment using virtual machines.

As a related technique to support the adaptive monitoring, *discovery* is a well-known useful technique to find a newly attached device in the network [9]. NNM provides the discovery function by collecting Address Resolution Protocol (ARP) tables in the target network. If a new server is connected to the target network, the monitoring tool supporting discovery can detect this new target. Although the detection of the new target is automated by discovery, the appropriate configurations for monitoring are up to the administrators. The administrators have to categorize the detected target and set the appropriate monitoring schedule not to have an adverse impact on the existing system.

Our adaptive monitoring system focuses on the quality of the monitoring service, specifically, information freshness and load of monitoring server. Appropriate configurations for monitoring server are important to maintain the quality of monitoring. Freshness is one of the important metrics for quality of resource monitoring [2]. If a monitoring interval is set to a large value, the data stored in the monitoring server is not up to date. The elapsed time from data generation exceeds the required time to live (TTL) and it causes the freshness degradation. To keep the freshness in the required level is important for monitoring aware applications and middleware. The stale (i.e. not fresh) information may cause the incorrect decision and control of monitoring aware applications. The load of the monitoring servers is another quality concern of monitoring systems. Monitoring processes consume system resources such as CPU time and network bandwidth. Excessive processes for information collection in a short time adversely affects system components sharing system resources as well as monitoring server. The processes for information collection need to be scheduled not to gather in a short time period.

3. Adaptive Monitoring System

In this section, we describe an architecture of an adaptive monitoring system and an algorithm for polling schedule generation.

3.1. Architecture

We designed an adaptive monitoring architecture based on the Web Service Polling Engine (WSPE) [2] that is a resource information service for server clusters. WSPE collects resource information from target server nodes via web service protocols, store the information into the temporal cache, and provide the information to the cluster users through the query interface. To keep the fresh information in the cache, WSPE updates the cache repeatedly as per the predefined update schedule. We improved this architecture to reconfigure the update schedule dynamically adapting to the system reconfigurations.

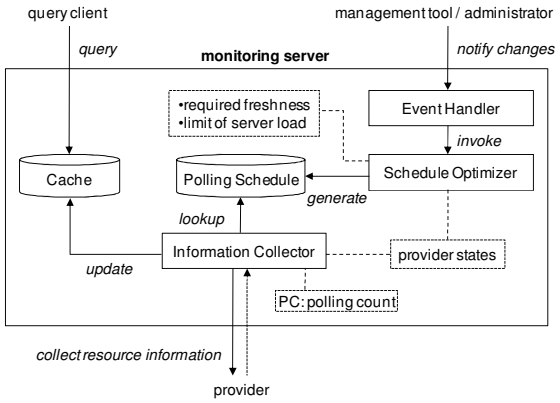

Figure 1. Adaptive monitoring system architecture

Figure1 shows an overview of the proposed monitoring architecture. The monitoring server consists of *Information Collector*, *Schedule Optimizer*, *Event Handler*, *Cache* and *Polling Schedule*. The *Information Collector* collects resource information from providers running on the target servers and updates the *Cache* with collected resource information. All queries from clients are performed on the *Cache*. The availability of each provider is also checked in the information collection process and is managed as provider states. An unavailable resource is dropped from the polling targets. The *Information Collector* counts the *Polling Count* (PC) that indicates the number of occurrences of polling cycles from the start-up. The target information that needs to be updated in one polling cycle is specified in the *Polling Schedule*. The *Polling Schedule* is determined so as to keep the freshness of resource information in the cache and the limit of server load. Since an optimum *Polling Schedule* is changed by the configuration and availabilities of target systems, the *Schedule Optimizer* calculates an optimum *Polling Schedule* in adapting to the latest system configurations. The trigger of schedule optimization is handled by *Event Handler* that receives several notifications about system reconfigurations from management middleware or administrators and determines the needs for schedule optimization. When the *Schedule Optimizer* receives a request for schedule optimization, it identifies the latest system configurations and generates a new *Polling Schedule* by a schedule optimization algorithm that is described in the later section.

The *Polling Schedule* is specified by the Next PC and the Interval PC for each resource as shown in Figure 2. The Next PC specifies the next PC at which to update resource information. When the PC in the *Information Collector* reaches a value of a Next PC, the Information Collector adds this target to the polling targets and collects the latest resource information from the provider. In order to reduce the risk of unexpected peak load caused by polling processes, dispersed values should be used for the Next PCs of different resources. If a large number of target resources have the same value of Next PC, the next polling process has to collect a large amount of resource information in one polling cycle and it may induce a heavy workload on the monitoring server. On the other hand, the Interval PC specifies the number of polling cycles between two consecutive updates. After a polling process to update resource information finishes, the value of the Next PC is calculated by adding the previous value of the Next PC to the Interval PC. The smaller value of Interval PC is preferable to keep the required freshness. The optimum Interval PCs are determined in consideration of the tradeoff between the required freshness and monitoring server load.

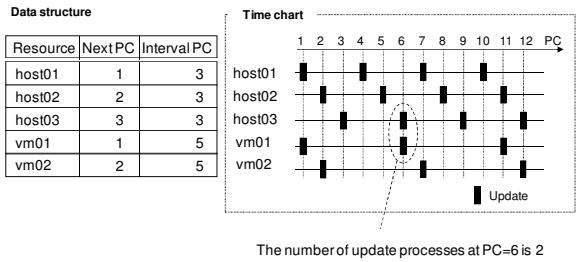

Figure 2. An example of update schedule

The max number of update processes in one cycle of polling must be limited to a certain range of values in consideration of the peak load of the monitoring server. Unexpected peak load called *flush peak* sometimes causes serious system trouble. Since the load of monitoring server depends on the number of target resources having the same Next PC, the peak load of the monitoring server is predictable by the *Polling Schedule* in the proposed system. By optimizing the *Polling Schedule* to keep the number of updates in one

polling cycle in a certain level, we can avoid the risk of the flush peak.

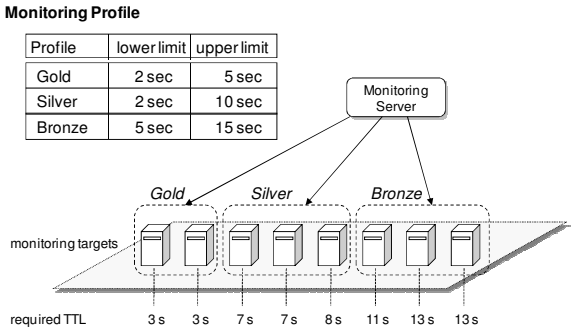

Figure 3. Monitoring profile to group resources

The *monitoring profile* figured in Figure 3 is introduced for grouping the target resources that have the same class of quality level. As the quality of the resource information, the freshness is specified by the TTL in detail. TTL indicates the elapsed time from data generation. The monitoring profile defines the lower limit and the upper limit of the update interval. Since a monitoring profile corresponds to a specific quality level, system administrator create a new monitoring profile when a new quality level is required. Each resource is assigned a monitoring profile and does not belong to the multiple monitoring profiles. Administrators simply manage the allocation of each resource to the specific monitoring profile instead of editing TTL for each resource. By using monitoring profile, the operation for the target addition and the change of monitoring frequency becomes much easier.

3.2. Schedule Generation Problem

The method to generate an optimal polling schedule is an essential part of the adaptive monitoring system. The polling schedule has to satisfy the required freshness of resource information and minimize the number of concurrent updates.

First, the Interval PC for each resource r_i is decided by the allocated monitoring profile p and the current polling interval t_{poll}. The minimum integer j that satisfies the limits defined in the profile is chosen as Interval PC. The Interval PC is expressed as the following expression:

$$\text{IntervalPC}(r_i) = \min\{\, j \mid j \in \mathbf{N}, \text{LL}_p \le t_{\text{poll}} \cdot j \le \text{UL}_p \,\} \tag{1}$$

where LL_p is the lower limit of the update interval for monitoring profile p and UL_p is the upper limit of that.

If any possible values are not found, the administrator should modify the monitoring profile or the polling interval to get a possible Interval PC. Meanwhile, the limited number of concurrent updates (LCU) in a polling cycle is decided in consideration to the acceptable load of the monitoring server.

Next, the Next PC for each resource is decided so that the number of the concurrent updates is not over the LCU. The number of the concurrent updates is changed by each PC and the way to set the Next PC. Since the update processes are executed repeatedly according to each Interval PC, the change in the number of the concurrent updates appears with a period of the least common multiple of Interval PCs (LCMI). We define the polling schedule generation problem as follows.

Problem: Polling Schedule Generation

For each resource information r_i, the update interval PC is defined as $\text{IntervalPC}(r_i) \in \mathbf{N}$. Solve the $\text{NextPC}(r_i) \in \mathbf{N}$ for all r_i, so that the number of concurrent updates is under the LCU at any k from 1 to LCMI.

Solve: $\forall i, \text{NextPC}(r_i)$

Where:

$$\forall k\ (1 \le k \le \text{LCMI}), \quad \sum_{i=1}^{n} \text{U}(k, r_i) \le \text{LCU} \tag{2}$$

$$\text{U}(k, r_i) = \begin{cases} 1 & k - \text{NextPC}(r_i) \equiv 0 \pmod{\text{IntervalPC}(r_i)} \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

$$1 \le \text{NextPC}(r_i) \le \text{IntervalPC}(r_i) \tag{4}$$

The schedule generation problem is an integer programming of $\text{NextPC}(r_i)$, that is classified as NP-hard. It takes exponential time of the number of targets "n" to decide if any possible schedule exists or not. If there are a large number of targets in the system, the above problem cannot be solved in practical time.

3.3. Schedule Generation Algorithm

To solve the schedule generation problem in practical time, we propose an algorithm by using an approximate method.

Algorithm 1:

1) Make groups that have the same value of $\text{IntervalPC}(r_i)$.

$$G_j = \{r_i \mid \text{IntervalPC}(r_i) = j\} \tag{5}$$

Define J as a set of possible values as j.

2) For each group, generate schedule that minimizes the concurrent updates. Label all r_i in G_j as $r_{i,k}$ $(1 \le k \le |G_j|)$ and set the NextPC($r_{i,k}$) based on this label.

$$\text{NextPC}(r_{i,k}) = k \ (\text{mod IntervalPC}(r_i)) \quad (6)$$

The number of max concurrent updates for G_j is calculated by: $\left\lceil \frac{|G_j|}{j} \right\rceil$

3) Combine all generated schedules and calculate sum of the number of concurrent updates.

$$\sum_{j \in J} \left\lceil \frac{|G_j|}{j} \right\rceil \quad (7)$$

Compare the sum of the number of concurrent updates to the LCU. If the sum of the number of concurrent updates is smaller than LCU, output the generated schedule as a possible schedule. Otherwise, give up the schedule generation.

Algorithm 1 divides the all r_i into the groups that have the same value of IntervalPC(r_i) and solves the partial optimal schedule for each group. By gathering the partial schedules, the max number of concurrent updates is minimized in most situations. Furthermore, the algorithm always outputs a result in $O(n)$ time.

If each pair of IntervalPC(r_i)s of the different groups is relatively prime, the Algorithm 1 always solves the optimal schedule (i.e. minimize the number of the concurrent updates) by the following theorems.

Theorem 1:

When all of the IntervalPC(r_i) have the same value, the max number of the concurrent updates of the schedule is equal to or more than $\left\lceil \frac{n}{\text{IntervalPC}(r_i)} \right\rceil$, where n is the number of targets.

Proof 1:

Let α be the max number of the concurrent updates. All of r_i have to be updated during IntervalPC(r_i) within α update processes.

$$n \le \alpha \cdot \text{IntervalPC}(r_i) \quad (8)$$

Because α is an integer value, the following condition is obtained.

$$\alpha \ge \left\lceil \frac{n}{\text{IntervalPC}(r_i)} \right\rceil \quad (9)$$

■

Theorem 2:

G_p and G_q are groups of resource information that has intervals of p and q. If p is coprime to q, the max number of the concurrent updates of the update schedule for all elements of G_p and G_q is equal to or more than $\left\lceil \frac{|G_p|}{p} \right\rceil + \left\lceil \frac{|G_q|}{q} \right\rceil$.

Proof 2:

For any $r_{p1} \in G_p$ and any $r_{q1} \in G_q$, the PC to update: $t_p(r_{p1})$ and $t_q(r_{q1})$ are generally represented by:

$$t_p(r_{p1}) = m_p \cdot p + \text{NextPC}(r_{p1}) \quad (10)$$

$$t_q(r_{q1}) = m_q \cdot q + \text{NextPC}(r_{q1}) \quad (11)$$

where, m_p and m_q are any positive integer values.

Here, for any NextPC(r_{p1}) and any NextPC(r_{q1}), there exists a pair of m_p and m_q satisfying $t_p(r_{p1}) = t_q(r_{q1})$ modulo pq. This is derived from the *Chinese remainder theorem* [6].

Therefore, there exists a case where the number of concurrent updates is 2 for any pair of r_{p1} and r_{q1}. The max number of the concurrent updates, α, is given by:

$$\alpha = \alpha_p + \alpha_q \quad (12)$$

where α_p and α_q are the max number of the concurrent updates for G_p and G_q.

From the Theorem 1, the following condition is obtained.

$$\alpha \ge \left\lceil \frac{|G_p|}{p} \right\rceil + \left\lceil \frac{|G_q|}{q} \right\rceil \quad (13)$$

■

Because the max number of the concurrent updates of the schedule generated by the Algorithm 1 is $\sum_{j \in J} \left\lceil \frac{|G_j|}{j} \right\rceil$, the output schedule is always optimal if each pair of IntervalPC(r_i)s of the different groups is relatively prime.

4. Evaluation

This section describes the experimental evaluations of the proposed adaptive monitoring system using a system reconfiguration scenario.

4.1. Monitoring load estimation

The load of the monitoring server such as CPU usage and the amount of the network traffic depends on the number of concurrent update processes. By investigating the relationship between the load of the monitoring server and the number of the concurrent updates, the load of the monitoring server at real

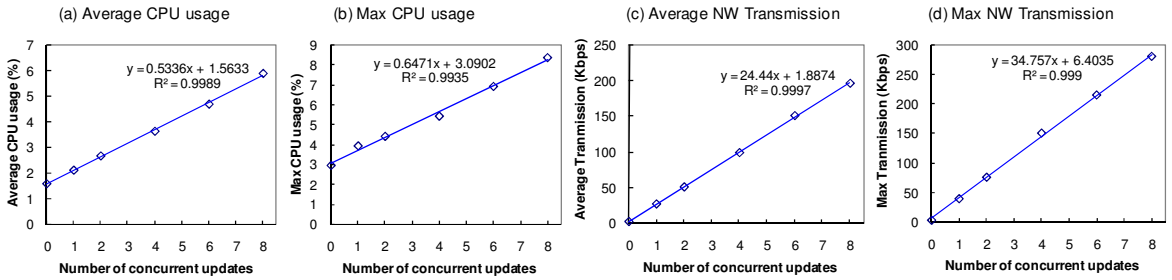

Figure 4. The relationship between the number of concurrent updates and the monitoring loads

execution can be estimated from the installed polling schedule.

The experimental environment has a monitoring server that has 3GHz Intel Pentium4 processor and 2.3 GB of RAM. On this server, WSPE collects resource information from several physical and virtual machines. Each target provides 12 KB of resource information. All nodes used in the experiments are connected by 100 Mbps ethernet.

In this testing environment, we measured several system metrics like CPU usages, memory usages, disk I/O and network traffics by varying the number of concurrent updates. The relationship between the system metrics and the number of concurrent updates can be characterized by regression analysis. Figure 4 (a) shows the plots of the measured values of CPU usages under the limited number of the concurrent updates. The relationship is expressed as the following expression by applying the least square method to the observed values.

$$y = 0.5336 \cdot x + 1.5633 \qquad (14)$$

where x is the number of concurrent updates in a polling cycle and y is the average CPU usage. The regression coefficients change depending on the resource capacities and states of usage. For example, the more CPU power the monitoring server can use, the smaller value the gradient of the regression line for the CPU usage. As far as this experimental environment is used, the average CPU usage of the monitoring server is predictable by the obtained regression formula. In addition to the average CPU usage, the max CPU usage, the average and max network transmission traffic also have the linear relation with the number of concurrent updates (see Figure 4). The other performance data such as network receive traffic, memory usage and disk I/O does not have linear relationship with the number of concurrent updates in our testing environment. From the results of this investigation, we can find an appropriate value of the number of concurrent updates to keep the load of monitoring server in a certain level.

4.2. Adaptation to system reconfigurations

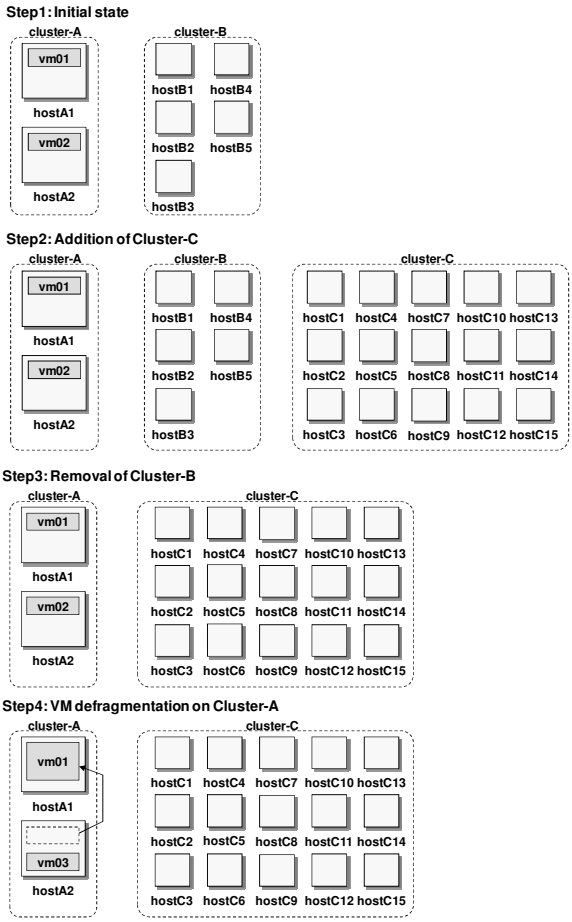

Figure 5. VM defragmentation scenario

The monitoring adaptation mechanism was evaluated by a scenario involving the virtual machine defragmentation as depicted in Figure 5. The monitoring setting is automatically changed by the proposed adaptation mechanism for each step of the

scenario. The experimental environment consists of three different clusters, cluster-A, cluster-B and cluster-C. The cluster-A is established on the virtualized environment using Xen 2.0 on Fedora Core 4. Cluster-B consists of 5 nodes and Cluster-C has 15 nodes.

In the first step of the scenario (step 1), the cluster-A and the cluster-B are monitored from the monitoring server running on a management server. In the second step (step 2), the cluster-C is added to the monitored target of the monitoring server. In the third step (step 3), the cluster-B is removed from the monitored target. In the final step (step 4), the defragmentation of virtual machines on the cluster-A is performed. The defragmentation moves the virtual machine instance vm02 to the hostA1, then merges instances of vm01 and vm02, and finally starts a new virtual machine instance vm03 in the created resource space on the hostA2. In this experiment, the merge process simply stops the vm02 and expands the resource allocation to vm01.

All physical servers and virtual machines have corresponding monitoring profiles. Table 1 shows the four different monitoring profiles used in the experiments. The polling interval t_{poll} is set to 1 second and the value of LCU is set to 8. For each step of the scenario, the optimization algorithm generates the optimal update schedule that meets the conditions specified in monitoring profiles and minimizes the number of concurrent updates under LCU. The generated update schedules for each step are shown in Table 2.

Besides the *optimization* approach, the *simple polling* approach and the *without-optimization* approach were also evaluated by this scenario for the sake of comparison. The simple polling approach updates all of information at regular intervals like SNMP polling. The regular interval was set to 10 seconds. The without-optimization approach updates resource information at specific intervals requested from each monitoring profile. Although this approach satisfies the conditions of the monitoring profiles, the number of concurrent updates is not bounded.

Table 1. Monitoring profiles

	lower limit	upper limit
Platinum	3s	10s
Gold	5s	15s
Silver	7s	20s
Bronze	11s	30s

Table 2. Update schedules for each step

cluster	node	profile	Interval PC	Next PC			
				Step 1	Step 2	Step 3	Step 4
A	hostA1	Platinum	3	1	1	1	1
	hostA2	Platinum	3	2	2	2	2
	vm01	Bronze	11	1	1	1	1
	vm02	Bronze	11	2	2	2	
	vm03	Bronze	11				2
B	hostB1	Platinum	3	3	3		
	hostB2	Platinum	3	1	1		
	hostB3	Platinum	3	2	2		
	hostB4	Platinum	3	3	3		
	hostB5	Platinum	3	1	1		
C	hostC1	Gold	5		1	1	1
	hostC2	Gold	5		2	2	2
	hostC3	Gold	5		3	3	3
	hostC4	Gold	5		4	4	4
	hostC5	Gold	5		5	5	5
	hostC6	Gold	5		1	1	1
	hostC7	Gold	5		2	2	2
	hostC8	Gold	5		3	3	3
	hostC9	Gold	5		4	4	4
	hostC10	Gold	5		5	5	5
	hostC11	Silver	7		1	1	1
	hostC12	Silver	7		2	2	2
	hostC13	Silver	7		3	3	3
	hostC14	Silver	7		4	4	4
	hostC15	Silver	7		5	5	5

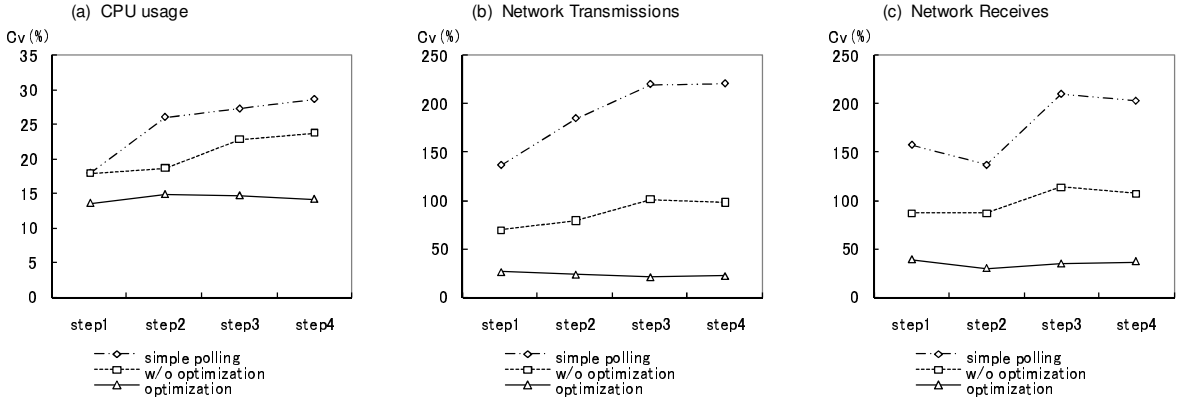

Figure 6. Variation coefficients of CPU usages and network traffics

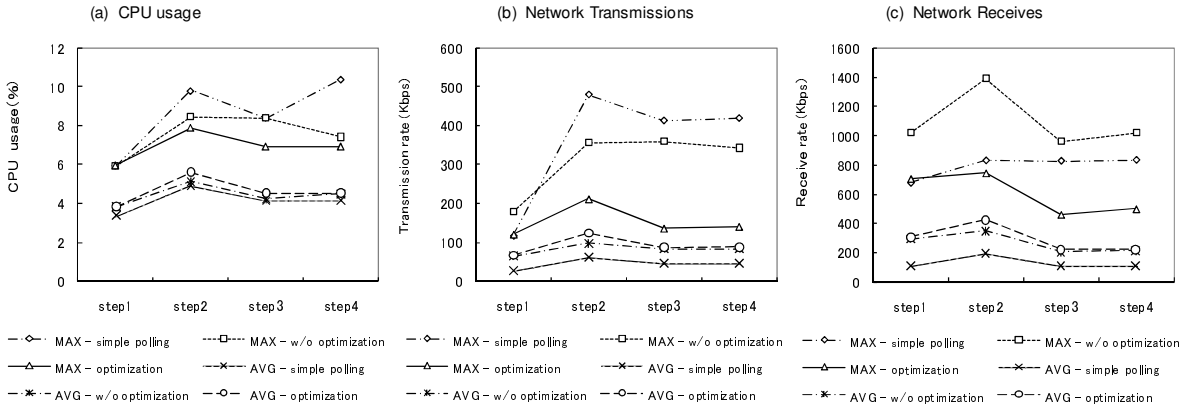

Figure 7. Max and average values of CPU usages and network traffics

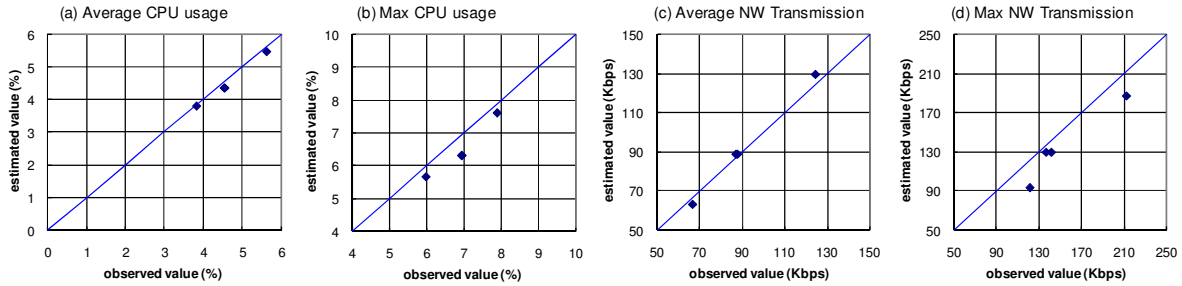

Figure 8. Observed values versus estimated values by regression functions

We observed the variation coefficients of CPU usages and network traffics for each step of the scenario (see Figure 6). All these variation coefficients were calculated from the time-series performance data of three minutes duration in each step. The variation coefficient of optimization approach is the lowest in any case and the values do not change significantly over the steps. Compared to the without-optimization approach, the variation coefficient of network transmission traffic is reduced by 80% at step 3 (see Figure 6 (b)). The results indicate that the proposed adaptive monitoring system stabilize the load of monitoring server by optimizing the polling schedule according to the system reconfigurations.

Meanwhile the max values of CPU usages and network traffics during the three minutes for each step are shown in Figure 7. The results provide a study of risk for flash peak of the resource usage. The optimization approach can lower down the max values of CPU usages and the network traffics by dispersing the update processes over time. Compared to the without-optimization approach, the max transmission traffic is reduced by 62% at step3 (see Figure 7 (b)). The proposed optimization approach reduces the risk of the flash peak.

Additionally the approximate max values are predicted by using the regression function described in Section 4.1. Figure 8 shows the relationship between the measured values and estimated values. The results show that the estimation provides a good indicator for availability of the monitoring server.

5. Scalable adaptive monitoring

In this section, we extend the adaptive monitoring system to hierarchical configurations. To satisfy the requirements for TTLs from lots of clients, we propose an algorithm for multiple schedules generation.

5.1. Requirements for scalable monitoring

Large scale enterprise systems distributed in multiple locations have thousands of monitoring targets such as servers, routers, switches and applications. A single monitoring server is not enough to collect the resource information from thousands of monitoring targets from the concern for the load of monitoring

server and network. Generally, for such a large-scale system, multiple monitoring servers are configured hierarchically to integrate the resource information. HP's NNM can manage 25000 of devices by organizing monitoring servers hierarchically. MDS [15] and Ganglia [16] support hierarchical architecture to aggregate resource information from thousands of nodes in the grid environment.

Although the hierarchically architecture improves the scalability of monitoring systems, the overhead of multiple monitoring servers degrades the query performance and freshness of resource information. Users and applications using the monitored information require the specific level of the query performance and information freshness. Configurations for monitoring servers for satisfying the quality requirements are much more complex than the case with a single server. Adaptive monitoring that reduces the manual operations for monitoring settings after system reconfiguration is also valuable in the large scale enterprise systems.

For the large scale enterprise systems, we extend the WSPE to hierarchical configurations. Each WSPE handles the event of system reconfigurations and adapts the polling schedule automatically to the target systems. By optimizing the polling schedule in each WSPE, all of requirements are satisfied under the capacity limitations of monitoring servers.

5.2. Hierarchical configuration of WSPEs

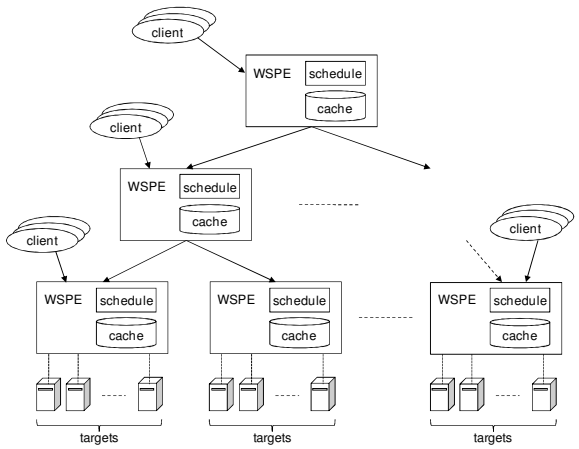

Figure 9. Hierarchically-configured WSPEs

Figure 9 shows a hierarchical configuration of WSPEs to collect resource information from widely-distributed systems. Each WSPE has own polling schedule to keep the freshness of the resource information in the cache. Some WSPEs collect resource information from the other WSPE instead of collecting directly from the target resources. It reduces traffics to the target resources and distributes the load of monitoring servers. Clients query the resource information to the nearest WSPE that has the target information in the cache. The query response time is estimated by the turnaround time from the client to the nearest WSPE.

The TTL of resource information r_i in the query results depends on the polling intervals of all WSPEs on the path from the client c_h to the target resource r_i. Here we denote the polling interval for resource r_i in the WSPE w_j as $t_{\text{poll}}(w_j, r_i)$. Let $W_{h,i}$ be the set of WSPEs on the path from the client c_h to the target resoruce r_i. The TTL of resource information r_i for the client c_h is bounded as the following expression:

$$t_{\text{TTL}}(c_h, r_i) \le t_{\text{resp}}(c_h, w^1, r_i) + \sum_{w_j \in W_{h,i}} t_{\text{poll}}(w_j, r_i) \qquad (15)$$

where $t_{\text{resp}}(c_h, w^1, r_i)$ is the time taken to deliver the information r_i from the nearest WSPE $w^1 \in W_{h,i}$ (see Figure 10).

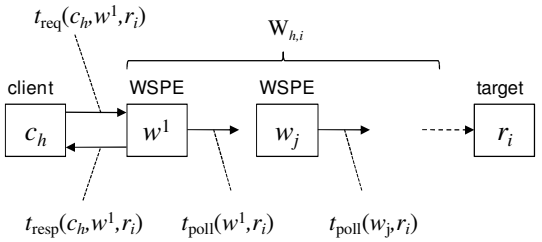

Figure 10. Model of hierarchical WSPEs

Let $t_{\text{req}}(c_h, w^1, r_i)$ be the time taken to request the query for r_i from c_h to w^1. The query response time is expressed as follows:

$$t_{\text{query}}(c_h, w^1, r_i) = t_{\text{req}}(c_h, w^1, r_i) + t_{\text{resp}}(c_h, w^1, r_i). \qquad (16)$$

If the $t_{\text{query}}(c_h, w^1, r_i)$ does not meet the required performance of c_h due to the limitations of network performance or server capacity, an additional placement of a WSPE near the client improves the query performance at the expense of the information freshness.

We assume the number of WSPEs and networks are given by the requirements for the query response time of each client and the limitation derived from the network topology. We discuss the problem of polling schedule optimization to guarantee the required TTLs for all clients under the limitations of server loads.

5.3. Multiple Polling Schedules Generation

Polling schedules for all WSPEs need to be optimized for satisfying the requirements for TTL of

resource r_i from the client c_h: RTTL(c_h, r_i) under the limitation of server loads given by the LCU of each WSPE. For a single WSPE, the Interval PCs are determined by the formula (1) based on the monitoring profiles. However, for the hierarchically-configured WSPEs where many clients request to guarantee the TTL of resource information, the Interval PCs need to be determined by considering the requested RTTLs and polling intervals of other WSPEs.

The problem to solve the polling schedules of WSPEs under the conditions about RTTLs and LCUs is defined as follows.

Problem: Multiple Polling Schedules Generation

Solve the IntervalPC(w_j, r_i) and NextPC(w_j, r_i) for each WSPE w_j to satisfy all requirements of RTTL(c_h, r_i), under the limitations of the number of concurrent updates LCU(w_j).

$$\text{Solve: } \forall i, \forall j, \quad \text{IntervalPC}(w_j, r_i), \text{NextPC}(w_j, r_i)$$

Where:

$$\forall h, \quad t_{\text{TTL}}(c_h, r_i) \le \text{RTTL}(c_h, r_i) \tag{17}$$

$$\forall k, \quad \sum_{i=1}^{n} \text{U}(k, w_j, r_i) \le \text{LCU}(w_j) \tag{18}$$

$$\text{U}(k, w_j, r_i) = \begin{cases} 1 & k - \text{NextPC}(w_j, r_i) \equiv 0 \ (\text{mod IntervalPC}(w_j, r_i)) \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

$$1 \le \text{NextPC}(w_j, r_i) \le \text{IntervalPC}(w_j, r_i) \tag{20}$$

Constraint (17) states the limitation from the requirements for RTTL(c_h, r_i). Constraints (18) and (19) state the limitation of the LCUs. The problem of multiple polling schedules generation is an integer programming and NP-hard as well as the schedule generation problem discussed in Section 3.2.

5.4. Multiple Schedules Generation Algorithm

We propose an algorithm to generate multiple polling schedules satisfying the requirements of RTTLs and the limitations of LCUs for hierarchically-configured WSPEs. The proposed algorithm generates polling schedules satisfying the constraints (18) by applying algorithm 1 for each WSPE and readjusts the Interval PCs so as to satisfy the constraints (17) by changing the assignment of monitoring profiles.

Algorithm 2:

1) Generate polling schedules for all w_j by applying algorithm 1 with the default monitoring profiles and the limitation of LCU that are set in each WSPE.
2) For all requirements for TTL of resource r_i from the client c_h, check if the max value of $t_{\text{TTL}}(c_h, r_i)$ calculated by (15) is below the RTTL(c_h, r_i). If all RTTLs are satisfied, output the schedules and finish the schedule generation process. Otherwise, go to the following steps to readjust the polling schedules.
3) Let w^k $(1 \le k \le |\text{W}_{h,i}|)$ be the sequence of WSPEs on the path to the r_i from c_h. The sequence starts from w^1 that is the nearest WSPE from c_h. In the sequence, search a w^k that can readjust schedule so as to satisfy the requirements of RTTL(c_h, r_i) by the following step 4. If the w^k that can readjust schedule is not found by the iteration of step 4, give up the multiple schedule generation.
4) In the given w^k, for resource r_i, change the allocation of profile that satisfies both of the following conditions.

$$\text{LL}_p \le \text{RTTL}(c_h, r_i) - t_{\text{TTL}}(c_h, r_i) + t_{\text{poll}}(r_i)$$
$$\text{UL}_p \ge \text{RTTL}(c_h, r_i) - t_{\text{TTL}}(c_h, r_i) + t_{\text{poll}}(r_i) \tag{21}$$

where LL_p is the lower limit of the update interval for monitoring profile p and UL_p is the upper limit of that. If any profile p that satisfies the conditions (21), calculate a new IntervalPC(w_j, r_i) by the expression (1) with the new profile and generate a schedule by the algorithm 1. Repeat finding the possible profiles until get the schedule or check all profiles.

Since the algorithm 2 is an approximation algorithm, it does not always output the multiple polling schedules even if there is a possible solution. However, the algorithm can change the polling schedules locally to satisfy the requirements of RTTL(c_h, r_i) instead of globally optimization. The algorithm gives the advantage to adapt the existing polling schedules to the change of RTTL(c_h, r_i). Since the monitoring profiles are edited by system administrator as necessary, the number of monitoring profiles is limited. The routine of step 4 is processed in the finite execution time.

6. Related work

Scalable performance monitoring systems have been well studied in the context of grid computing. A white paper summarized and evaluated lots of presented grid monitoring systems [13]. Some advanced monitoring systems such as Remos [11] and Network Weather Service (NWS) [12] have a function

to forecast the performance changes. In contrast to several existing works for the grid monitoring systems, we focus on the quality of the monitoring service, namely freshness of resource information, in the large scale enterprise systems.

The quality of monitoring is important especially in the grid and autonomic computing. The monitoring requirements differ across applications hosted on the server and change over time corresponding to the system configurations. QMON [4] provides a function to classify and configure the quality of monitoring based on service level agreement (SLA). QMON changes the monitoring configuration dynamically by using the concept of "monitoring channel". However, the current QMON does not support the adaptation mechanism to the target system reconfiguration such as server addition and deletion.

Although freshness is important for applications using monitored data, the significant emphasis on the freshness results in a "flash crowd" caused by monitoring processes [14]. The monitoring system must manage the server load to avoid the flash crowd. Our experimental results show that the flash crowd is avoidable by the optimized schedule.

For the network management, an efficient polling technique for SNMP is proposed [5]. This technique provides a function to minimize the polling queries to the SNMP agents by using the usage parameters defined by the applications. However, any method to avoid the flash crowd is not supported.

The necessity of the polling optimization is also described in the grid monitoring system using slacker coherence model [3]. The slacker coherence model is useful to minimize the polling with consideration to the out-of-sync period of the data. Although this model considers the load of the target nodes, the server-side load is not considered. Therefore, there is no guarantee that the flash crowd does not occur.

7. Conclusion

This paper proposed the adaptive monitoring system to reduce the administrative operations in the large-scale enterprise systems. The monitoring server guarantees the freshness of resource information in the cache by the polling based cache updates. The update processes are scheduled to satisfy the requirements of freshness and the limitation of monitoring server load. We presented a schedule generation algorithm and proved that the algorithm generates an optimal schedule minimizing the max number of concurrent updates. From the experimental results, the variation coefficients of CPU usages and network traffics are improved by at most 80%, and the max values at the load peak are decreased by at most 62%. The results show that the proposed method can stabilize the load of monitoring server and can reduce the risk of flash peak according to the current system configuration. We presented as well the extension of the adaptive monitoring system to be scalable with the algorithm for generating multiple polling schedules. By applying the proposed algorithm to hierarchically-configured WSPEs, we can guarantee all requirements for freshness of resource information from multiple users under the limited loads of monitoring servers.

References

[1] F. Machida, M. Kawato and Y. Maeno, Adaptive Monitoring for Virtual Machine Based Reconfigurable Enterprise Systems, 3rd International Conference on Autonomic and Autonomous Systems (ICAS2007), 2007.

[2] F. Machida, M. Kawato and Y. Maeno, Guarantee of Freshness in Resource Information Cache on WSPE: Web Service Polling Engine, 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid2006), 2006.

[3] R. Sundaresan, M. Lauria, T. Kurc, S. Parthasarathy and Joel Saltz, Adaptive Polling of Grid Resource Monitors Using a Slacker Coherence Model, 12th IEEE International Symposium on High Performance Distributed Computing (HPDC03), 2003.

[4] S. Agarwala, Y. Chen, D. Milojicic and K. Schwan, QMON: QoS- and Utility-Aware Monitoring in Enterprise systems, 3rd IEEE International Cofference on Autonomic Computing (ICAC2006), 2006.

[5] M. Cheikhrouhou and J. Labetoulle, An Efficient Polling Layer for SNMP, IEEE/IFIP Network Operations and Management Symposium (NOMS2000), 2000.

[6] D. E. Knuth, The Art of Computer Programming, Volume 2: Seminumerical Algorithms, 3rd Edition, Section 4.3.2, page 286, Addison-Wesley, 1997.

[7] B. Korte, J. Vygen, Combinatorial Optimization: Theory and Algorithms, Japanese Edition 2005, Section 15.7 NP-Hard Problems, Springer, 2005.

[8] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham and R. Neugebauer, Xen and the Art of Virtualization, 19th ACM Symposium on Operating Systems Principles (SOSP19), 2003.

[9] Y. Breitbart, M. Garofalakis, C. Martin, R. Rastogi, S. Seshadri, and A. Silberschatz, Topology discovery in heterogeneous IP networks, 19th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM2000), 2000.

[10] J. Case, M. Fedor, M. Schoffstall and J. Davin, "Simple Network Management Protocol (SNMP)", RFC 1157, 1990.

[11] P. Dinda, T. Gross, R. Karrer, B Lowekamp, N. Miller, P. Steenkiste, and D. Sutherland, The architecture of the remos system. In 10th IEEE International Symposium on High Performance Distributed Computing (HPDC-10), August, 2001.

[12] R. Wolski, Experiences with Predicting Resource Performance On-line in Computational Grid Settings, ACM SIGMETRICS Performance Evaluation Review, Volume 30, Number 4, March, 2003, pp 41--49.

[13] M. Gerndt, R. Wismueller, Z. Balaton, G. Gombas, P. Kacsuk, Z. Nemeth, N. Podhorzki, H. Truong, T. Fahringer, M. Bubak, E. Laure and T. Margalef. Performance Tools for the Grid: State of the Art and Future, Automatic Performance Analisis: Real Tools White Paper, 2004.

[14] R. Desai, S. Tilak, B. Gandhi, M. J. Lewis and N. B. Abu-Ghazaleh, Analysis of Query Maching Criteria and Resource Monitoring Models for Grid Application Scheduling, 6th IEEE International Symposium on Cluster Computing and the Grid (CCGrid2006), 2006.

[15] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman, Grid Information Services for Distributed Resource Sharing, In Tenth IEEE International Symposium on HighPerformance Distributed Computing (HPDC10), IEEE Press, August 2001.

[16] M. L. Massie, B. N. Chun, and D. E. Culler, The Ganglia Distributed Monitoring System: Design, Implementation, and Experience, Parallel Computing, Vol. 30, Issue 7, July, 2004.

[17] HP OpenView Network Node Manager (NNM): http://h20229.www2.hp.com/products/nnm/index.html

[18] IBM Tivoli NetView: http://www-306.ibm.com/software/tivoli/products/netview/

[19] ZABBIX: http://www.zabbix.org/

[20] OpenNMS: http://www.opennms.org/index.php/Main_Page

[21] Nagios: http://nagios.org/

[22] VMware: http://www.vmware.com/

[23] Microsoft Virtual Center: http://www.microsoft.com/windowsserversystem/virtualserver/