CloudMon: A Resource-Efficient IaaS Cloud Monitoring System Based on NIDS Virtual Appliances

Bo Li[1], Jianxin Li[1], Lu Liu[2]
[1]School of Computer Science and Engineering, Beihang University, Beijing, China
[2]School of Computing and Mathematics, University of Derby, Derby, UK
libo@act.buaa.edu.cn, lijx@act.buaa.edu.cn, l.liu@derby.ac.uk

ABSTRACT

The Networked Intrusion Detection System Virtual Appliance (*NIDS-VA*), also known as virtualized NIDS, plays an important role in the protection and safeguard of IaaS cloud environments. However, it is non-trivial to guarantee both of the performance of *NIDS-VA* and the resource efficiency of cloud applications since both are sharing computing resources in the same cloud environment. To overcome this challenge and trade-off, we propose a novel system, named *CloudMon*, which enables dynamic resource provision and live placement for *NIDS-VAs* in IaaS cloud environments. *CloudMon* provides two techniques to maintain high resource efficiency of IaaS cloud environments without degrading the performance of NIDS virtual appliances and other virtual machines. The first technique is a VMM-based resource provision mechanism which can minimize the resource usage of a *NIDS-VA* with given performance guarantee. It uses a fuzzy model to characterize the complex relationship between performance and resource demands of a *NIDS-VA*, and develops an online fuzzy controller to adaptively control the resource allocation for *NIDS-VAs* under varying network traffic. The second one is a global resource scheduling approach for optimizing the resource efficiency of the entire cloud environments. It leverages virtual machine migration to dynamically place *NIDS-VAs* and virtual machines (VM). An online VM mapping algorithm is designed to maximize the resource utilization of the entire cloud environment. Our VMM-based resource provision mechanism has been evaluated by conducting comprehensive experiments based on Xen hypervisor and *Snort* NIDS in a real cloud environment. The results show that the proposed mechanism can allocate resources for a *NIDS-VA* on demand, while still satisfying its performance requirements. We also verify the effectiveness of our global resource scheduling approach by comparing it with two classic vector packing algorithms and the results show that our approach improved the resource utilization of cloud environments and reduced the number of in-use *NIDS-VAs* and physical hosts.

KEY WORDS: cloud environments, NIDS virtual appliance, fuzzy control, resource management, dynamic provision

1. INTRODUCTION

With the rapid development of Internet and networking technologies, the emerging Internet-of-Things (IoT) [1] is expected to be a future interconnection environment that connects computers, electronic devices and all other things that can be uniquely identifiable and linked to the Internet. Cloud computing, as a new resource delivery and consumption model, can provide the virtual computing and storage infrastructure for IoT environments to integrate various devices and process the big data generated by them, which has become the key enabling technology of IoT. Recently, some IoT-oriented cloud systems and services [2][3][4][5] are emerging and show great potential for future development.

However, on the other side, due to the open and dynamic natures of cloud computing, several security issues arise and pose great challenges to the application of cloud systems in the IoT domain. In IaaS

Cloud environments, traditional security systems like Networked Intrusion Detection Systems (NIDS) cease to be effective in such kind of "virtualized" environments. Firstly, traditional NIDS cannot monitor and analyze the intra-VM traffic (the network traffic occurs among the virtual machines hosted on the same physical server), because communications between these VMs never reach the physical network. Secondly, server consolidation produces more traffic than traditional non-virtualized environments, which significantly increases the burden of NIDS. Lastly, network virtualization creates an overlay topology upon physical networks, and the effectiveness and efficiency of traditional NIDS will be suffered due to the unawareness of the existence of overlay networks.

NIDS Virtual Appliance [6] (also known as virtualized NIDS, or vIDS) is an effective way to address the above issues. Physical NIDS devices are virtualized and encapsulated into virtual machines which can be easily deployed into physical hosts in cloud environments and monitor the internal traffic between VMs without the need for exporting the traffic outside of the physical host. Besides, NIDS Virtual Appliances allow users to consolidate and manage NIDS systems and devices in a virtualized way which can, therefore, greatly reduce hardware costs and simplify IT management. However, it is non-trivial for NIDS Virtual Appliances deployment in IaaS environments. We identify three key challenges in the deployment and resource management of NIDS virtual appliances as follows.

The first challenge is how to deploy a NIDS virtual appliance into a physical server without affecting the performance of other VMs hosted on the same physical server. In virtualized environments, physical resources are shared among VMs, which are often consolidated for the efficient use of server resources. The sharing of computing resources will result in resource competition between NIDS virtual appliance and the other VMs running on the same physical server. The resource competition will affect both the performances of workload VMs and the detection accuracy of NIDS virtual appliance. In order to guarantee the performance of virtualized NIDS, a common-used approach is allocating enough resources to the NIDS virtual appliance according to its maximum resource demand. However, the NIDS may not always work at its full load, which leads to resource idleness and waste thereby violating the objective of server consolidation. This, therefore, reflects a tradeoff between resource utilization and performance of VMs in general and NIDS in particular. To improve resource efficiency without sacrificing the performance of NIDS, one way could be establishing a precise mathematical model to characterize the relationship between workloads and resource requirement of NIDS virtual appliance. Unfortunately, the complex nature of NIDS poses great challenges to accomplish this objective.

The second challenge is how to place NIDS Virtual Appliances and VMs to cover all the network traffic sent to or received from the workload VMs, while still maximizing resource utilization of physical servers and minimizing the overall resource consumption of NIDS virtual appliances. For NIDS Virtual Appliances, since network traffic loss may lead to intrusion undetected, any traffic including internal and external traffic should be monitored by NIDS. To ensure that any intra-VM traffic is under inspection, an ideal approach is to deploy a NIDS Virtual Appliance for each workload VM, whereas this approach needs the same number of NIDS Virtual Appliance as the workload VMs and wastes lots of resources. Note that the operating system of NIDS virtual appliance will also occupy resources even when the NIDS application is idle. To reduce energy costs and maximize platform revenue, the server resources used to host workload VMs and the number of the deployed NIDS Virtual Appliances should be minimized. Therefore the key problem is to find an appropriate placement strategy of NIDS Virtual Appliances and workload VMs which can achieve optimal resource utilization.

Thirdly, in virtualized environments, NIDS often faces varying network traffic, and therefore, the resource consumption will fluctuate accordingly. Besides, the applications in workload VMs often have varying resource demands, e.g., an application may request more resources during its running, or else the service quality of the applications will be decreased dramatically. The dynamic characteristic of

NIDS virtual appliance and workload VMs will lead to resource competition or resource underutilized. VM live migration technique can be used to mitigate this problem by remapping VMs to different physical servers, so the key issue is how to derive an online placement algorithm which can dynamically adjust the placement scheme adapting to the varying resource demands of NIDS Virtual Appliances and workload VMs. Besides, for each adjustment, the times of live migration should be limited to a small extent.

To address the above issues, we propose *CloudMon*, a resource-efficient NIDS-based IaaS cloud monitoring system, which enables adaptive deployment and dynamic resource provision for NIDS virtual appliances. Our major contributions are summarized as follows:

- A novel NIDS dynamic provision approach is proposed based on fuzzy control theory, which can continuously adjust resource allocation for NIDS virtual appliance to deal with varying network traffic while still satisfying the performance requirements of NIDS.
- The VM placement problem is modeled as a vector bin packing problem, and an offline VM placement strategy is adopted to determine the most preferred placement scheme. Then we propose an event-driven online heuristic algorithm to derive the VM mapping scheme regarding the dynamic characteristic of NIDS virtual appliance and workload VMs.
- An online feedback-driven control system is developed to accurately adjust resource allocation according to the NIDS's performance requirement in a real-time manner. A prototype system is implemented in our iVIC platform. *Snort* NIDS [7] and Xen hypervisor are used to evaluate the effectiveness of our approach. Experimental results show that our feedback fuzzy control approach can effectively allocate resource to NIDS virtual appliance under time-varying network traffic while still satisfying the performance requirements of NIDS. The results also show the effective and efficiency of our VM placement and mapping algorithms.

The paper is organized as follows. We give a system overview of *CloudMon* in Section 2. Section 3 introduces our fuzzy control approach for adaptive resource allocation, and Section 4 presents the design of our cloud-oriented VM placement algorithms for global resource management. We introduce the implementation of *CloudMon* in Section 5. The performance evaluation is given and analyzed in Section 6. We discuss related work in Section 7. Finally, we conclude the whole paper in Section 8.

2. SYSTEM OVERVIEW

This section discusses the basic design principle and system architecture of *CloudMon*. The background and terminology of *CloudMon* is first introduced and defined in the following sub-sections.

2.1 Background and Terminology

An IaaS cloud environment is usually composed of hundreds or even thousands of physical hosts (physical servers). Each host consists of processor, memory, storage as well as network interfaces. For simplicity, we assume all the physical hosts are homogenous. The hosts are interconnected by high-speed Gigabyte LAN. Each host runs a virtual machine monitor (VMM) and hosts one or more VMs. A VM may communicate with either VMs inside of the cloud computing environment or the destinations outside. We use *internal traffic* to refer to the network traffic among VMs, and use *external traffic* to represent the network traffic going to the outside world. It is worth noting that *internal traffic* can still be divided into two types: *intra-VM traffic* and *inter-VM traffic*. *Intra-VM traffic* represents the network traffic among VMs which are located in the same physical host, thus will not go outside of the physical host; on the contrary, *inter-VM traffic* is transmitted by VMs hosted by different hosts, and this kind of network traffic will traverse the physical network interface and switch.

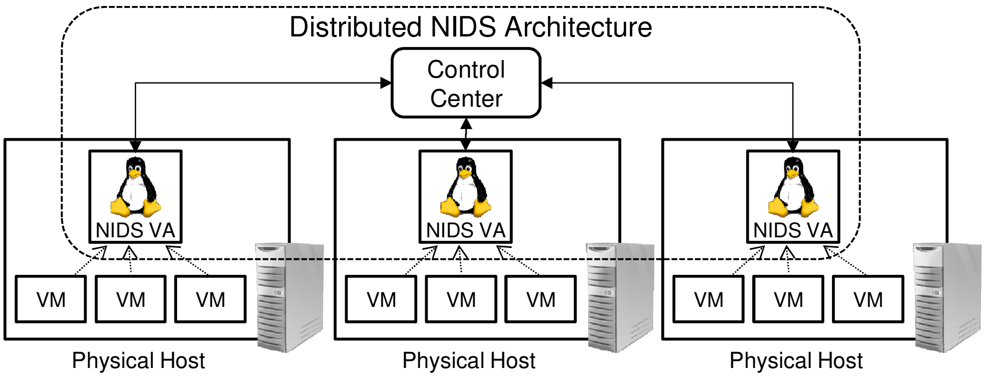


Figure 1: A Generic Cloud Monitoring Framework based on Distributed NIDS

Figure 1 shows a generic framework of a NIDS-based cloud security monitoring system which monitors the whole network traffic of the cloud environments, including *internal traffic* and *external traffic*. The NIDS software and the underlying OS are encapsulated into a VM which is referred to as NIDS virtual appliance (*NIDS-VA*). Each host is equipped with one or more *NIDS-VA*s. The *NIDS-VA* is responsible for inspecting incoming and outgoing network traffic of workload VMs hosted on the same physical host, and it shares the resources such as CPU and memory with other VMs. A virtual network (vNet) is used to connect the *NIDS-VA*s among different hosts and the control center, and to isolate the network traffic among *NIDS-VA*s from normal traffic among VMs. The intrusion detection results such as intrusion alerts and risk warnings generated by the *NIDS-VA*s are reported to the control center for taking actions and further analysis.

2.2 Architecture

CloudMon is designed based on the above generic monitoring framework, while it focuses on the performance and resource efficiency issues which have not been considered and discussed in existing systems. Besides the prerequisite that each VM, if needed, is under the monitor of at least one *NIDS-VA*, *CloudMon* has two distinguished design requirements: (1) Providing performance guarantees for both the *NIDS-VAs* and the workload VMs; (2) Maximizing the resource utilization of cloud environments monitored by distributed *NIDS-VAs*. To meet the two requirements, we propose the architecture of *CloudMon* (as shown in Figure 2) which can achieve resource efficiency of cloud environments without sacrificing the performance of both the *NIDS-VA*s and the workload VMs.

CloudMon is composed of two main components: *Local Resource Manager (LRM)* and *Global Resource Scheduler (GRS)*. *LRM* is an in-host dynamic resource manager which allocates resources for *NIDS-VAs* and workload VMs according to their resource requirements. The core of *LRM* is an online fuzzy controller which can adaptively control the resource allocation for *NIDS-VAs* under varying network traffic. It leverages fuzzy model to characterize the complex relationships between performance and resource consumption of NIDS, and implements a dynamic resource provision approach to efficiently provide resources to *NIDS-VA* while still fulfilling its performance demand. *GRS* aims at optimizing the resource utilization of the entire cloud. It has a global view over the resources of cloud environments, and uses a VM mapping algorithm to maximize the resource utilization of the entire cloud environments. *GRS* takes into account the resource usage of both *NIDS-VAs* and workload VMs. To deal with the varying of resource needs, an event-driven VM online mapping algorithm is presented to guide the placement and migration of VMs so as to mitigate system hotspots and maintain high resource utilization. *GRS* makes placement and migration decisions based on the resource utilization information reported by *LRM*, and *LRM* is responsible for executing the real VM operations like *start, stop* and *migrate* based on the decisions from *GRS*.

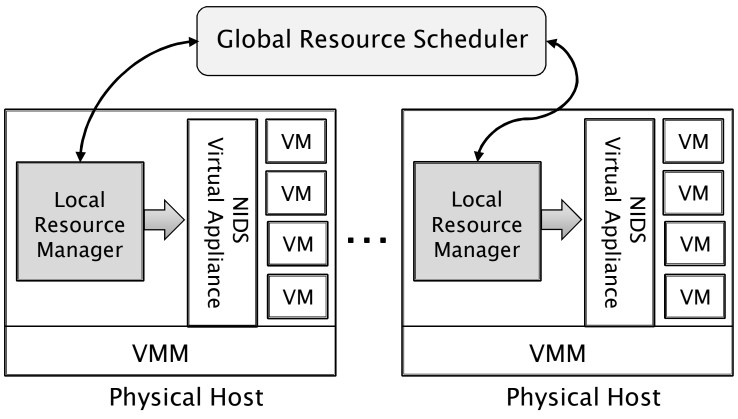


Figure 2: The Architecture of *CloudMon*

3. A DYNAMIC RESOURCE PROVISION APPROACH FOR *NIDS-VA*

For *NIDS-VA* deployment and resource management in an IaaS cloud environment, the first step is to deploy each *NIDS-VA* into a physical server and determine how much resource should be allocated to it. The successful deployment of a *NIDS-VA* needs an accurate resource provision approach and should satisfy two requirements: (1) allocating appropriate resource to it to avoid resource waste; (2) providing performance guarantees for both the *NIDS-VA* and the other VMs hosted on the same physical server.

3.1 Problem Statement and Analysis

For simplification, we only consider CPU and memory resources. In addition, we assume that I/O resources including disk and network I/O are sufficient. But it is important to note that the CPU cycles or memory pages consumed when handling I/O operations are considered here. Furthermore, we assume the CPU and memory allocation can be adjusted through VMM interface when the NIDS virtual appliance is running. It is reasonable since most VMMs such as Xen and VMware ESX server support run-time CPU and memory adjustment.

In this section, we focus on investigating how resource provision will affect the performance of NIDS. With respect to CPU and memory allocation, *packet drop rate* is the most relevant performance metric. Therefore, we choose it as the performance indicator. In the rest of this paper, *packet drop rate* and *performance* of NIDS are used interchangeably. We use *DR* to represent the packet drop rate. Since no NIDS can guarantee zero packet loss, we define a target drop rate T*DR*. If *DR<= TDR*, we conclude the performance of NIDS to be satisfactory.

If the resource allocation is too small, NIDS will experience performance degradation and allow malicious packets to enter the network undetected. However, if the NIDS is over-provisioned, it will lead to poor resource utilization and resource waste. There is a trade-off between performance and resource requirements, especially for IaaS environments in which *NIDS-VA* shares physical resources with other VMs. In order to elaborate this tradeoff, we give the following example.

Example - *Consider such a scenario where one NIDS virtual appliance and one VM are located on the same physical host. We assume TDR to be 0.02 i.e. the acceptable drop rate is 2%. Now, we assume the VM needs 50% CPU to fulfill the performance requirements of the application it contains. But the NIDS virtual appliance currently occupies 60% CPU, while its drop rate is 1% or even lower. Therefore, there exists a resource competition so that the QoS requirement of the application cannot be fulfilled. However, if the drop rate of NIDS can be 1.8% when allocating 50% CPU to the NIDS, we can fulfill the requirements of both the NIDS and the VM. In other words, if we adjust the CPU allocation of NIDS to 50%, both the NIDS and the application can be satisfied. However, this requires knowledge of the relationship between resource demands and performance.*

From the above discussion, we can conclude that the key problem is to, given a network traffic rate, determine appropriate resources that need to be allocated to the *NIDS-VA* to satisfy the performance of *NIDS APP*. Figure 3 shows the model of NIDS Virtual Appliance.

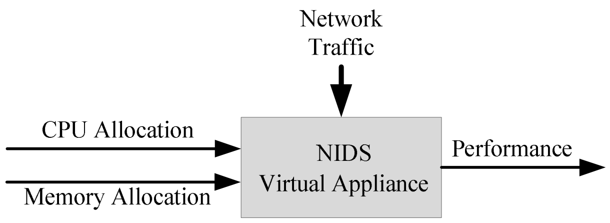


Figure 3: A Model of NIDS Virtual Appliance

The CPU allocation, memory allocation and network traffic affect the performance of NIDS in varying degrees as described below.

CPU: CPU overload directly leads to packet drops. A NIDS is a soft real-time system where processing needs to keep up with the input network traffic and failing to do so will result in packets being dropped. Therefore, it is very important to determine the CPU demands of NIDS, especially in a shared environment. Unfortunately, it is a non-trivial task. Firstly, the complex nature of detection engine makes it difficult to estimate its CPU demands. Detection engine is the most CPU consuming part, and it is responsible for analyzing network traffic based on detection rules. However, different rules will incur different processing time. For example, some rules only inspect the packets header, while others may deep into the payload for further analysis. On the other hand, it is very hard to estimate the overall resource consumption by the whole NIDS appliance. This is because the overall resource consumption includes the resource consumption done by *NIDS APP* as well as the *NIDS-VA*. For example, network I/O operations in OS kernel also consume CPU cycles.

Memory: Memory overload could also result in packet drops. Historically, a NIDS often consumes a large amount of memory pages. First, NIDS operates in a stateful fashion i.e. it needs to store connection states for further analysis. Second, when analyzing network traffic, NIDS requires memory to store the intermediate results. Last, but the most important, NIDS will buffer packets before analyzing them to keep up with the high traffic speed. In heavy traffic load conditions, the buffer is filled up quickly. When it is full, packets may drop. For the IDS virtual appliance, physical memory is first allocated to *NIDS-VA*, and when *NIDS APP* needs memory it will issue "*malloc*" to request memory from the OS of *NIDS-VA*. If the request cannot be satisfied, *NIDS APP* may crash. For modern operating systems which support virtual memory and swapping, the *NIDS APP* will not crash, but its performance will be greatly degraded due to frequent swap activities.

Network Traffic: The patterns of network traffic being analyzed influence the resource usage, therefore, it will also affect drop rate indirectly. First, a NIDS consumes different processing time for different protocols. A study [8] revealed that Snort NIDS spends about 30% processing time for pattern matching, while for web traffic the cost increased to 80%. Second, the proportion of suspicious packets affects the processing speed of NIDS. More resources are required for alerting and logging when dealing with suspicious packets. Last, heavy traffic load will incur packet loss of *NIDS-VA* when the resource provision is inadequate. The experimental results (Section 6.2) show that when CPU provision cannot fulfill the resource demand of network traffic, large amount of packets fail to be received by the kernel of *NIDS-VA*, much less being captured by *NIDS APP*.

Through the above analysis, we can conclude that NIDS's performance could be influenced by many factors. It is difficult to construct a precise mathematical model to characterize the correlation between performance and resource consumption, especially when handling a variety of network traffic.

3.2 A Fuzzy Controller

Instead of designing a mathematical model to characterize the correlation among network traffic load, resource requirements and performance of NIDS virtual appliance, we resort to feedback control techniques to adaptively allocate resources to meet its performance requirements. We have considered traditional feedback control systems, most of which require specifying the mathematical models in advance. However, NIDS virtual appliance is too complex to be represented by a mathematical model. To solve this problem, fuzzy models [3] have been used to characterize the complex relationship between performance and resource demands. Fuzzy logic is effective in dealing with the uncertain, imprecise, or qualitative decision-making problems, which has been successfully applied into control systems to handle problems that are too complex to be modeled by conventional mathematical methods. As a result, we designed a fuzzy logic-based controller which controls the resource allocation to the *NIDS-VA* based on a set of linguistic IF-THEN rules, thus it is not necessary to establish a mathematical model for the *NIDS-VA*. Our fuzzy controller meets the following three design requirements: (1) the performance of NIDS virtual appliance can be guaranteed; (2) appropriate resources are allocated to the *NIDS-VA* to avoid competing resources with other virtual servers; (3) high resource utilization could be maintained.

As shown in Figure 4, a fuzzy controller controls the resource allocation to the *NIDS-VA* according to the performance feedback of the *NIDS-VA*. The input of *NIDS-VA* is the resource allocation, and it should be dynamically adjusted to meet the resource requirements of time-varying network traffic workloads. The output of the *NIDS-VA* is the performance metric being measured, and it will be fed back to the fuzzy controller. As mentioned above, we choose the Drop Rate (DR) as the performance indicator since it is the most relevant metric regarding the resource allocation of an NIDS. The fuzzy controller takes drop rate deviations as input and outputs a control action *U* on how much resource will be allocated. Two kinds of drop rate deviations are considered here: *e* and *ec*. Those are:

$$e = \mathrm{CurrentDR} - \mathrm{TDR}$$
$$ec = \frac{e_t - e_{t-1}}{\mathrm{time\_interval}} \tag{1}$$

The control action will be executed by the VMM to adjust the actual resource allocation for the *NIDS-VA*. Figure 4 shows a feedback control loop, and the goal of our fuzzy controller is continuously adjusting the resource allocation to approximate the drop rate target.

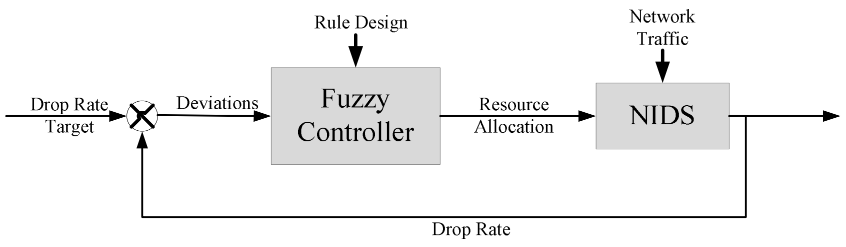


Figure 4: Fuzzy controller for dynamic resource allocation

The structure of our fuzzy controller is shown in Figure 5. The fuzzy control process is divided into three steps. The first step is *fuzzification* which maps the input variables into fuzzy sets by membership functions; the second step is *inference* which takes fuzzy sets as input and makes decisions for what action to take based on some verbal or linguistic rules of if-then form; the last step is *defuzzification* which translates the linguistic control actions into quantifiable outputs. Table 1 shows the details of our fuzzy controller.

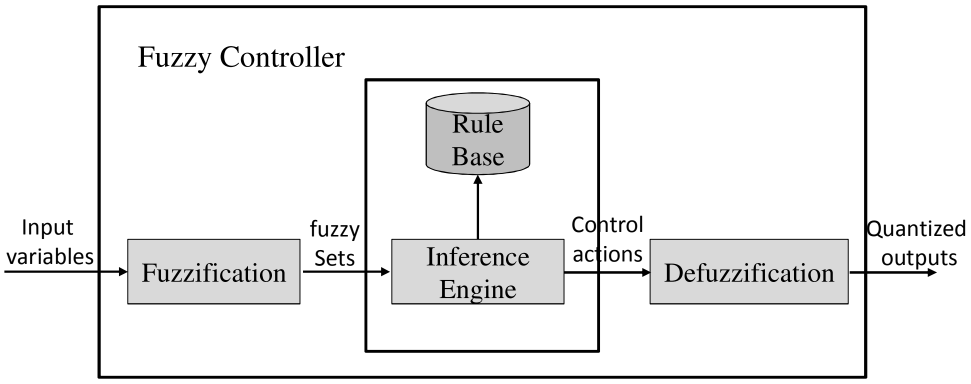
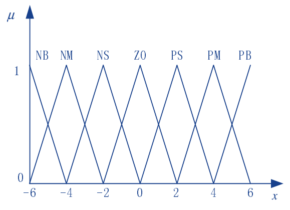


Figure 5: Building blocks of a fuzzy controller

Table 1. the details of our fuzzy controller

	Description
State variable	the input of the controller, here is *e* and *ec*
Control variable	the output of the controller, here is *U*
Fuzzy Set	The input and output are normalized on the interval [-1, +1]. The space of input and output is partitioned into seven regions. Each region is associated with a linguistic term. The membership function of the fuzzy sets is TRIANGULAR. $\mu_{\underset{\sim}{A_i}}(x) = \begin{cases} \frac{1}{b-a}(x-a), & a \le x < b \\ \frac{1}{b-c}(u-c), & b \le x \le c \\ 0 & \text{else} \end{cases}$ 
Fuzzification	We choose Singleton fuzzifier which measures the state variables without uncertainty.
Fuzzy Rule	We collect some expert knowledge through comprehensive experiments, and design the following rules:

U		EC						
		NB	NM	NS	Z	PS	PM	PB
E	NB	NB	NB	NB	NB	NM	Z	Z
	NM	NB	NB	NB	NB	NM	Z	Z
	NS	NM	NM	NM	NM	Z	PS	PS
	Z	NM	NM	NS	Z	PS	PM	PM
	PS	NS	NS	Z	PM	PM	PM	PM
	PM	Z	Z	PM	PB	PB	PB	PB
	PB	Z	Z	PM	PB	PB	PB	PB

Inference method	We choose the Mandani inference method since it can deal with both fuzzy input and output
Defuzzification	Center of gravity method is selected for defuzzification. $U^* = \left\langle \frac{\sum_i \mu_{\underset{\sim}{C}^*}(U_i)U_i}{\sum_i \mu_{\underset{\sim}{C}^*}(U_i)} \right\rangle$

4. MAPPING ALGORITHMS FOR OPTIMIZING THE RESOURCE EFFICIENCY OF CLOUD ENVIRONMENTS

In the previous section, a dynamic resource provision approach has been presented for each *NIDS-VA*, and in this section we are going to further investigate how to place the *NIDS-VAs* and workload VMs in a resource-efficient way while still fulfilling the performance requirements of both of them. Recall that a prerequisite for successfully monitoring a cloud computing environment is to cover all network traffic sent or received by each VM. An intuitive way is to equip each VM with a *NIDS-VA*. This method will result in large amounts of resources wasted, because both the operating system of *NIDS-VA* and the NIDS sensor software will occupy a certain amount of resources even in an idle state (idle state means no traffic is under detection). Thus, the more *NIDS-VAs* running, the more resources wasted. For a distributed NIDS system, to maximize its resource efficiency and minimize resource waste, the key is to minimize the number of *NIDS-VAs* used to cover the network traffic transferred by all the VMs. Besides, the overall resource utilization of the cloud environments should also be considered for cost saving and energy conservation reasons. Next, we will elaborate this problem.

4.1 Problem Analysis

The problem described above can be decomposed into two sub-problems: a Traffic-to-NIDS mapping problem and a VM-to-Host mapping problem.

A. Traffic-to-NIDS Mapping Problem

To monitor intra-VM traffic, at least one *NIDS-VA* should be deployed into each physical host. In order to minimize the number of *NIDS-VAs* and to simplify the problem, we assume that only one *NIDS-VA* is deployed for each physical host. The network traffic of all the VMs on this physical host is monitored by this *NIDS-VA*. The network traffic, which a *NIDS-VA* can process, is limited and *MaxCap* is used to represent its maximum capacity. Therefore, our problem is to determine an appropriate placement scheme to map a set of VMs' network throughputs to a minimum number of *NIDS-VAs* with the constraints that the sum of the network throughputs of the VMs should not exceed the *MaxCap* of the *NIDS-VA* for each physical host. This problem is a classic bin packing problem with NP-completeness.

B. VM-to-Host Mapping Problem

On the other side, in IaaS cloud environments, VMs are usually consolidated tightly to maximize resource utilization and minimize the number of online physical hosts for energy conservation and cost saving reasons. While a physical host is shared by the workload VMs and the *NIDS-VA* hosted on it, the sum of resource consumptions of both the workload VMs and the *NIDS-VA* should not exceed the host resource capacity. A VM-to-Host mapping schema is needed to minimize the number of online physical hosts without affecting the performance of both the workload VMs and the *NIDS-VA*s, with the constraint that for each host, the overall resource consumptions of both the hosted workload VMs and the *NIDS-VA* should not exceed the host's resource capacity. This problem is not a classic bin packing problem since the resource consumption of *NIDS-VA* is related with the network traffic it monitors. Figure 6 illustrates an example with six VMs and two *NIDS-VA*s. The number in the center of red box represents the resource consumption of a *NIDS-VA*. In Figure 6 we can see that the resource consumption of a *NIDS-VA* varies with the changing of network traffic under monitor. While in classic bin packing problems, the packing items are invariable.

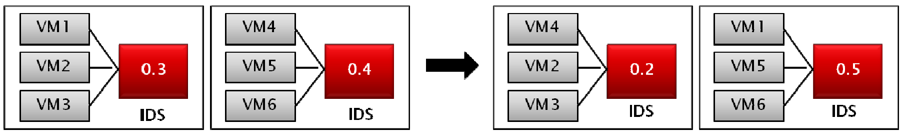


Figure 6: an example with six VMs and two *NIDS-VA*s

To tackle this problem, we first break down the resource consumption of a *NIDS-VA* into two parts: (1) $\text{Resource}_{\text{idle}}$: the resource used to run an operating system and an idle NIDS; (2) $\text{Resource}_{\text{load}}$: the resource consumption caused by NIDS workloads (network traffic). We adopt the model proposed by [17] which hypothesizes that the resource consumption of a NIDS is *orthogonal decomposition* which estimates aggregate resource requirements as the sum of the individual requirements. Based on this hypothesis, the overall resource consumption of a *NIDS-VA* can be broken down into two parts: (1) $\text{Resource}_{\text{idle}}$: the resource used to run an operating system and an idle NIDS; (2) the sum of resource consumptions caused by each VM's network traffic. Figure 7 gives a simple example. VM_1 represents the resource consumption of this VM, while R_{vm1} represents the resource consumption of *NIDS-VA* caused by VM_1, and we treat VM_1 and R_{vm1} as a whole and then consider putting them on which host. The problem described above can be transformed to a classic bin packing problem: the bin size is the host capacity minus the resource consumption of idle NIDS, and the items to be put into the bins are the VMs' resource consumption plus the resource consumption of NIDS caused by the traffic workload of this VM. We assume that the resource consumptions of VMs are known beforehand and we use the fuzzy control method proposed in Section 3 to derive how much resource should be allocated to a *NIDS-VA* for dealing with each VM's network traffic.

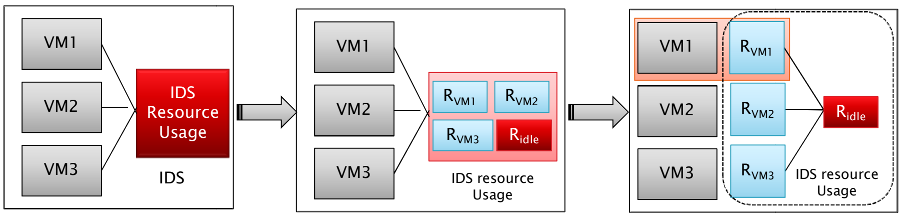


Figure 7: Breakdown of NIDS Resource Usage

The optimization objective of traffic-to-NIDS mapping problem is to minimize the number of *NIDS-VA*s, while the objective of VM-to-Host mapping problem is to minimize the number of physical hosts. In fact, the two objectives are consistent, since we assume that only one *NIDS-VA* is deployed for each physical host. We combine the two problems into a single-objective constrained optimization problem which is also known as a multi-dimensional vector packing problem (MVPP) [20]. MVPP is a variant of multi-dimensional bin packing problem (MBPP). The difference between them is that dimensions are independent for MVPP, but dependent for MBPP dimensions. In the one-dimensional case, MVPP and MBPP are identical. The classical bin packing was one of the original NP-complete problems [21]. MVPP is a generalization of the classical one-dimensional bin packing problem, and it is clearly NP-complete.

4.2 An Offline Mapping Algorithm

An Virtual environment is modeled as a set P={1,2, …, m} of physical hosts indexed by *l* and a set of NIDS={1,2, …, m} indexed by *i*, with finite capacity C_p and C_{ids}, respectively. They are shared by a set VM={1,2, …, n} of virtual machines indexed by *j*. We assume all the physical hosts are homogenous. Associated with each VM_i are CPU consumption VM_i^c, Memory consumption VM_i^m and network traffic rate VM_i^{nt}. We use $NIDS_c$ and $NIDS_m$ to denote the CPU and memory consumption of NIDS in idle state, respectively. We define:

$$P_l = \begin{cases} 1 & used \\ 0 & unused \end{cases}$$

$$A_{jl} = \begin{cases} 1 & VM_j \text{->} P_l \\ 0 & otherwise \end{cases}$$

$$A_{ji} = \begin{cases} 1 & VM_j \text{ monitored by } IDS_i \\ 0 & otherwise \end{cases}$$

For any j, if $i=l$, then $A_{jl}=A_{ji}$ since the network traffic of a VM can only be monitored by NIDS hosted on the same physical host. Because each VM can only be put into one host, so we have:

$$\sum_{l=1}^{m} A_{jl} = 1 \text{ and } \sum_{i=1}^{m} A_{ji} = 1$$

Then we formula the problem as an Integer Linear Program (ILP) problem:

$$\text{Minimize: } \sum_{l} P_l \qquad (1)$$

$$\text{Such that } \sum_{l=1}^{m} A_{jl} = 1 \,, \quad 1 \le j \le n \qquad (2)$$

$$NIDS_c + \sum_{j=1}^{n} A_{jl} \times VM_j^c \le C_p^n \,, \quad 1 \le l \le m \qquad (3)$$

$$NIDS_m + \sum_{j=1}^{n} A_{ji} \times VM_j^m \le C_p^m \,, \quad 1 \le i \le m \qquad (4)$$

$$\sum_{j=1}^{n} A_{jl} \times VM_j^{nt} \le C_{ids} \,, \quad 1 \le l \le m \qquad (5)$$

$$A_{jl} \in (0,1), 1 \le j \le n, 1 \le l \le m \qquad (6)$$

Constraint (2) states that every VM should be placed into a physical host. Constraint (3) and (4) ensures that the NIDS's idle resource consumption plus the resource consumptions of hosted VMs do not exceed the host's resource capacity. Constraint (5) ensures that the sum of the network traffic rate of each VM does not exceed the capacity of *NIDS-VA*. Constraint (6) shows that a VM can be either entirely put into a host or not.

To solve the above integer linear program (ILP) problem, we adopt an efficient mathematical algorithm which is proposed by C. S. Rao et al [28] based on the combination of (near-)optimal solution of the Linear Programming (LP) relaxation and a greedy (modified first-fit) heuristic. In [28], the authors provide a 2-OPT guarantee for large inputs irrespective of the dimension *d*. This is a notable improvement over the previously known guarantee of $\ln(d + 1 + \varepsilon)$ for any $\varepsilon \ge 0$ and higher dimensions $d > 2$. Due to space limitation, we will not present the details of the algorithm.

4.3 An Online Mapping Algorithm

Our discussion thus far has assumed that the resource consumptions of VMs and *NIDS-VA*s at run-time do not change after initial phase. However, in a real cloud environment, the resource requirements of VMs are usually changing dynamically. As for *NIDS-VA*, its resource utilization varies with the changes of network traffic workloads. The inflation of VMs' resource requirement will lead to resource competition, which may affect the performance of both workload VMs and the *NIDS-VA* hosted on the same physical host; the deflation of VMs' resource requirement will result in resource idleness and low utilization. Besides, VMs generally arrive and depart dynamically, which aggravates the above problem. Thus, a dynamic resource allocation approach is needed which can allocate resources based on the real-time resource requirements of VMs with a variety of workloads and an online VM mapping algorithm which can dynamically adjust the VM-to-Host mappings to deal with resource competition, mitigate system overload and maintain high resource utilization. The dynamic resource allocation of *NIDS-VA* is implemented using our fuzzy-control-based resource provision approach proposed in Section 3. As for the dynamic resource allocation of application VMs, research work [29] has given a feasible solution, and our dynamic resource provision approach is also an option. Next, we will elaborate the model and design of our VM mapping algorithm.

4.3.1 An Event-Driven Model

There are four kinds of events which can trigger VM remapping. The first is VM arrival. To minimize the number of online hosts, our algorithm should try to put the new arriving VM into an existing online host without using a new one. The placement of a new arriving VM will affect the *NIDS-VA* located on the same host in two aspects: (1) the network traffic of the arriving VM increases the workloads of *NIDS-VA*, and a feasible placement should ensure that all the traffic workloads of *NIDS-VA* do not exceed its maximum capacity; (2) the resource requirements of *NIDS-VA* will be enlarged due to the increasing of network traffic workloads caused by the arriving VM. The second is VM departure which may release some resources and lead to resource underutilized. Our algorithm should try to vacate an online host and hibernate it to save costs and maintain high utilization. The third is the inflation of VM's resource requirements which may result in system overhead and performance degradation. The last is the deflation of VM's resource requirements which may lead to low utilization. *NIDS-VA* is a special type of VMs. Its resource inflation and deflation should also be considered. Just like the first two events, our algorithm should try to relocate the VMs to use the minimal online hosts while still satisfying VMs' performance requirements.

The working model of our algorithm is illustrated in Figure 8. The algorithm handles event sequence in an online manner, and output a sequence of plans. Upon receiving each event, it generates a VM remapping plan including a series of VM placement and migration operations which guides the remapping of VMs.

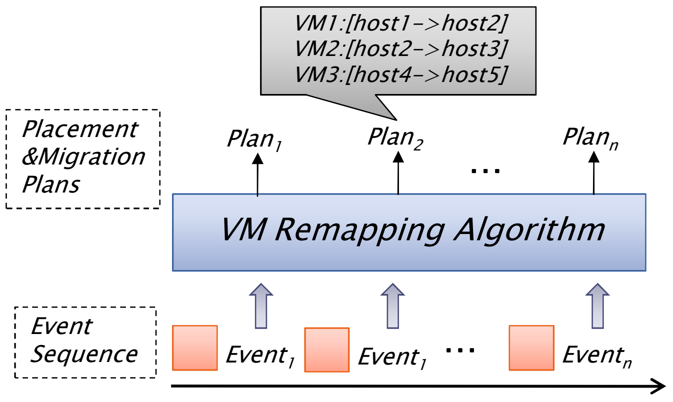


Figure 8: Working Model of Online VM Mapping Algorithm

4.3.2 Algorithm Design

The problem described above shows some similarities with the online vector packing problem [22]. However, there are two main differences between the two problems. First, in our problem the packing item's size (resource consumptions of a VM) is changing over time, while the item's size is fixed for online vector packing. Second, no migration of items is allowed for online vector packing problem. Because of such differences, algorithms suited to online vector packing problem cannot be applied to our problem.

We design a heuristic VM mapping algorithm based on the idea proposed in our previous work [30]. The basic idea is that the smaller items are recursively replaced by bigger items until all items are packed. It is based on an obvious observation the smaller items are easier to be inserted into the gaps of bins. Our contribution in this paper is to redesign our previous algorithm and extend it from one-dimension to multi-dimension.

In order to reduce the number of reinsertions and the complexity of the algorithm, we divide the region of the VM size into several subintervals, and each subinterval represents a *level*. A modified Best-Fit algorithm, Best-Fit-3D', is employed to pack the newly arrived VM. Best-Fit-3D' differs from traditional Best-Fit algorithm in two aspects. First, when putting a new VM_i using Best-Fit-3D', the VMs whose levels are lower than VM_i will be temporarily ignored. After putting VM_i, the VMs hosted on the same host whose levels are lower than *level of* VM_i, are extruded in ascending order until the host is packable. A question is that how to compare the levels of two VM vectors since two dimensions may have an opposite comparative result. Second, for multi-dimensional vector packing problems, different dimensions may have different best-fit degrees, and a unified metric is required to characterize the degrees among multiple dimensions. To answer the above questions, two definitions are given.

Definition 1 For two multi-dimensional vectors, VM_1 and VM_2, the level of VM_1 is higher than the level of VM_2 if: (1) for each dimension the level of VM_1 is higher than or equal with the level of VM_2; (2) at least in one dimension the level of VM_1 is higher than the level of VM_2.

Based on Definition 1, it can be easily derived that the algorithm is convergent since the algorithm will eventually stop when the VMs with the smallest level are placed into the hosts. It's worth noting that the number of VM levels will directly influence the migration times of VMs. Next, we define a unified metric for measuring the fitness of best-fit for multi-dimensional vector packing.

Definition 2 For a host with n dimensional resources, $Host_i$ represents the resource utilization of the ith dimension. The fitness function of Best-Fit' is defined as follows:

$$Fitness = \prod_{i=1}^{n} \frac{1}{1 - Host_i}$$

Our Online-VM-Mapping algorithm is described as follows:

Online-VM-Mapping Algorithm

```
Variables:
  VM set: {$x_1$, $x_2$, ..., $x_n$ }  indexed by i
  Host set: {$P_1$, $P_2$, ..., $P_n$}  indexed by j
Procedure: Put(x)
    if min_level(x)= level_max || max_level(x)=level_min
        return First-Fit-3D(x)
    fi
    p*=Best-Fit-3D'(x)
    sort each VM x* in p* to { $x_1$*, ... , $x_n$* } in ascending order
    for  i =1 to n
        if packable($x_i$*)
            break
        fi
        pop( $x_i$*)
            Insert ($x_i$*)
    endfor
EndP
Procedure: Relocate(x)
    Pop(x)
    Put(x)
EndP
Procedure: Main
    While(true)
        Switch(event)
            Case VM_Arrive:
             Put($x_i$)
            Case VM_Depart:
                if under-utilized($p_j$)
            foreach VM $x_i$ in host $p_j$
                    relocate($x_i$)
          endeach
                fi
          Case VM_Inflation:
             While overloaded($p_j$)
                   relocate($x_{min}$)
             endWhile
          Case VM_Deflation:
             if under-utilized($p_j$)
         foreach VM $x_i$ in host $p_j$
                   relocate($x_i$)
        endeach
           fi
       endSwitch
    endWhile
EndP
```

To avoid the overhead of frequent VM migrations due to the varying of VMs' resource demands, two thresholds are defined in our algorithm: *under-utilized* and *overloaded*. Only when the resource utilization of a host is below *under-utilized* or above *overloaded*, a VM remapping event will be triggered.

5. IMPLEMENTATION

In this section, we will present the implementation details of *CloudMon*, show some technical details of the *NIDS-VA* resource manager, and introduce our implementation experience in iVIC finally.

5.1 CloudMon

As mentioned before, *CloudMon* is composed of two main components: *Global Resource Scheduler (GRS)* and *Local Resource Manager (LRM).* As shown in Figure 9, *GRS* runs the *Information Service (IS), VM Mapping Manager (VMMA)* and *Event and Job Dispatcher (EJD). IS* periodically receives the resource usage information (such as resource utilization, etc.) of VMs and hosts from *Resource Monitor (RM)* module located in *LRM*, this information will first be stored and then be sent to *EJD. EJD* receives the events from users (e.g. a user submit an application VM to run) and *IS* (e.g. a VM has to be remapping due to the overload of a host) and delivers them to *VMMA*. *VMMA* is responsible for generating VM placement and migration plans. It can work in two modes: offline and online. When working at the offline mode, a VM mapping plan will be generated based on the resource utilization information stored in *IS*. In the online mode, *VMMA* receives the VM arrival, VM departure and resource change events sent by *EJD*. It will generate VM mapping scheme composed of a series of VM placement and migration operations. The scheme will be sent back to *EJD*. *EJD* decomposes the scheme to a series of placement and migration commands, and then dispatches them to the *VM Manager* of *LRM*. Upon receiving these commands, *VM Manager* will invoke the VM management interface of the hypervisor to execute them. *NIDS-VA Manager* is the core module of *LRM.* It controls the resource allocation of *NIDS-VA*, which is composed of *fuzzy controller, NIDS performance monitor* and *allocation actuator*. This will be explained in detail in Section 5.2.

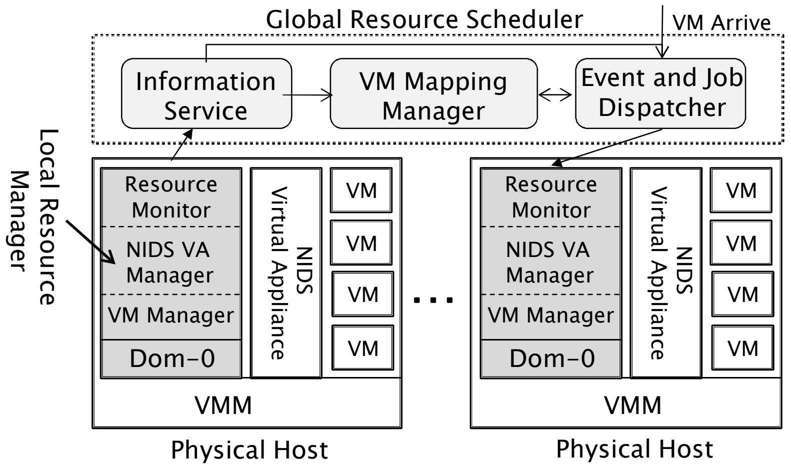


Figure 9: The Implementation of *CloudMon*

We implement our global resource management framework using Python. The offline and online VM mapping algorithms are implemented in *VMMA*. The communication between *Control Panel* and local services is based on SOAP protocol. In the *VM Manager*, an adaptive provision method [29] has been adapted to adjust the resource allocation of VM during runtime. Xen's non-work conserving credit scheduler is also employed by the *VM Manager* to control CPU allocation. The VM management function in *VM Manager* is implemented using Xen's Python management API to start, stop and migrate VMs. By querying `xentop`, the *Resource Monitor* obtains real-time performance statistics of each VM, which includes CPU and Memory usage.

5.2 The Fuzzy-Control-Based *NIDS-VA* Resource Manager

Currently, *NIDS-VA Resource Manager* is working on Xen hypervisor, we choose `Debian lenny` as an operating system for dom0, NIDS virtual appliance and workload VMs. We use snort NIDS, since it is the most widely used open source network intrusion detection system. The system architecture is illustrated in Figure 10.

In Xen, domain 0 is a special privileged domain which serves as an administrative interface to Xen. As shown in Figure 10, three modules of *NIDS-VA Resource Manager* are in domain 0. The *performance sensor* module periodically collects drop rate statistics from the *performance monitor* plug-in of NIDS APP, and sends drop rate data to the *fuzzy controller* module. The *fuzzy controller* consists of four components. First, the *fuzzifier* will map the drop rate data into some fuzzy values by given membership functions. Then, the inference engine will infer from the fuzzy values to make decisions and produce the output actions according to the fuzzy rules stored in the *Rule base*, the output actions is also fuzzy values. Last, the *defuzzifier* will combine the output values to give a crisp value, the actual resource allocation for the NIDS virtual appliance. The *allocation actuator* will execute the resource allocation actions made by the fuzzy controller through Xen interface. It is worth noting that the *performance sensor* and *allocation actuator* communicate with the *fuzzy controller* using the TCP protocol, which means the *fuzzy controller* can be located at any place as long as it has network connection with other modules, thus achieves good scalability. The enforcement of new resource allocation made by the *allocation actuator* will impact the performance of *NIDS Software* inside the *NIDS-VA*, and the *performance monitor* component of *NIDS Software* will record the drop rate data and send them to the *Performance sensor* through *XenStore* channel. It is the end of a control loop, and a new one will begin.

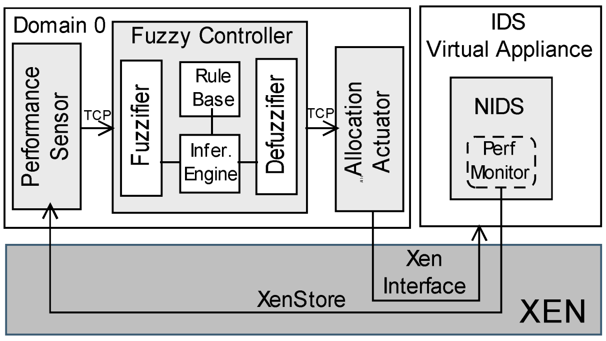


Figure 10: Fuzzy-Control-based Resource Provision Architecture

The *performance sensor* is implemented as a Linux daemon. Besides periodically collecting drop rate data, it also gathers real-time performance statistics of the NIDS virtual appliance through *xentop*, such as CPU usage and network throughout. The *Fuzzy controller* is implemented in C programming language in consideration of performance. The *TRIANGULAR* membership function is employed to map the input into fuzzy values. The inference engine is implemented using the *Mandani* inference method. Fuzzy rules, together with state variable, control variable and fuzzy sets, are stored in a text file, and loaded into the inference engine when the fuzzy controller is started. The *allocation actuator* is responsible for enforcing the CPU and memory adjustment decisions. For CPU scheduling, Xen credit scheduler has been chosen in non-working conserving mode, which provides strong performance isolation. Memory ballooning mechanism is adopted for run-time memory allocation adjustment. The *performance monitor* plug-in of snort is modified to transmit the real-time performance data outside of the virtual appliance. It is undesirable to use network for data transmission, since network transmission will disturb the detection of NIDS. An efficient performance data transmission mechanism is added into snort which leverages the *XenStore* mechanism to exchange information between domain 0 and NIDS virtual appliance without interrupting the network.

5.3 Implementation in iVIC

The prototype of *CloudMon* has been implemented in iVIC[1] cloud platform. IVIC is a typical IaaS cloud environment which enables users to dynamically create and manage various kinds of virtual resources such as virtual machines, virtual clusters and virtual networks. As a basic security service of iVIC, *CloudMon* can detect network attacks and intrusions and protect virtual machines and virtual networks from being attacked and intruded.

As shown in Figure 11, users can easily deploy *NIDS-VA*s into iVIC environments through drag-and-drop operations. In Figure 11, an *NIDS-VA* is deployed into a host machine, and monitors a VM[2] hosted on that machine. We implement a user-friendly interface using ruby-on-rails. Users can start or cancel the monitor of a given VM through dragging it on or off the *NIDS-VA*. *CloudMon* supports exporting the network traffic of a VM outside of the host machine, which enables the *NIDS-VA* to monitor the network traffic of a VM on another host.

CloudMon has been integrated into the implementation of iVIC. Figure 12 shows a detailed information page of a virtual network. This page is refreshed every 5 seconds, and users can monitor the

[1] Http://www.ivic.org.cn
[2] In CloudMon, a VM being monitored by a NIDS-VA is shown in a list below the icon of the NIDS-VA.

security status of a virtual network and its VM nodes in a real-time manner. As shown in Figure 12, the two VM items which are marked as red indicate that the two VMs are under attacks. *CloudMon* is able to detect 220 types of attacks and has obtained about 40,000 attacks samples.

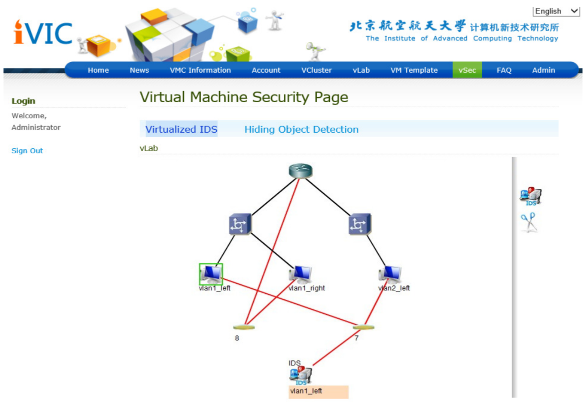
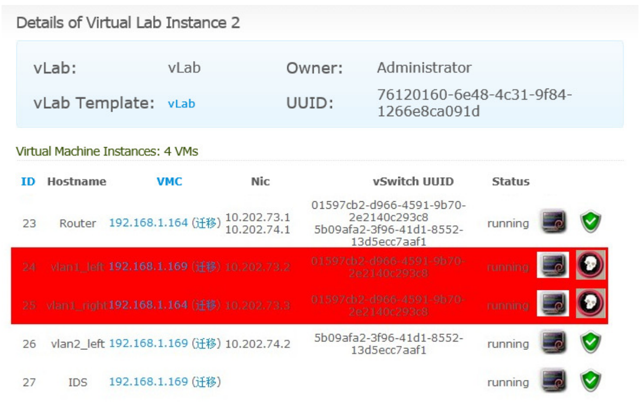


Figure 11: ***NIDS-VA*** **Deployment in iVIC**



Details of Virtual Lab Instance 2

vLab:	vLab	Owner:	Administrator
vLab Template:	vLab	UUID:	76120160-6e48-4c31-9f84-1266e8ca091d

Virtual Machine Instances: 4 VMs

ID	Hostname	VMC	Nic	vSwitch UUID	Status		
23	Router	192.168.1.164 (迁移)	10.202.73.1 10.202.74.1	01597cb2-d966-4591-9b70-2e2140c293c8 5b09afa2-3f96-41d1-8552-13d5ecc7aaf1	running		
24	vlan1_left	192.168.1.169 (迁移)	10.202.73.2	01597cb2-d966-4591-9b70-2e2140c293c8	running		
25	vlan1_right	192.168.1.164 (迁移)	10.202.73.3	01597cb2-d966-4591-9b70-2e2140c293c8	running		
26	vlan2_left	192.168.1.169 (迁移)	10.202.74.2	5b09afa2-3f96-41d1-8552-13d5ecc7aaf1	running		
27	IDS	192.168.1.169 (迁移)			running		

Figure 12: Attacks Being Identified by ***CloudMon***

6. EXPERIMENTAL EVALUATION

In this section the experiment configuration will be introduced and experiment results will be analyzed and discussed.

6.1 Experiment Setup

A series of experiments have been conducted to evaluate the effectiveness of our approaches. Our experimental environment is based on an iVIC cloud pool consisting of 64 physical servers with Intel Core2 Quad 2.83 GHz, 8G RAM, Linux 2.6.26 operation system, Xen 3.2 hypervisor, and gigabit Ethernet connection. We use one physical server to run the global control panel, and seven of them are used as clients to send requests to server application. An NIDS virtual machine image is prepared which encapsulates our modified version of snort 2.7 and mounts a 2GB disk image as the primary partition and a 1GB image for swap partition.

NIDS Workloads Generation. Network traffic traces have been collected to test the performance of Snort. *Tcpdump* is used to capture and save the network packets into a `.pcap` trace file. *Tcpreplay* is used to resend the captured packets from the trace file, and it also provides the function to control the speed at which the traffic is replayed. In order to impose various loads on Snort, we collect various kinds of network traffic traces. For example, we capture normal network traffic traversing the gateway of our lab; we also use some tools such as *nessus*, *nmap* and *snot* to generate malicious packets and then capture them using *tcpdump*.

VM workloads Generation. Three types of VM workloads have been selected to simulate the diversity of applications in a cloud environment. For compute-intensive applications, we use Spec CPU 2006 [31] to generate different levels of CPU and Memory loads. We choose tcpdreplay, netpipe and BitTorrent [23] as a test case for network applications. By controlling their packet sending rates, we can generate various types of network traffic. We also choose some commonly-used applications such as Kernel Compile, Apache Web server and MySQL server. Each application is encapsulated into a VM which will start automatically with VM boot. Note that the execution time of these applications is different, and we choose them to simulate various workload lifetime. For example, Apache Web server and Database Server are long-running workloads, while the workload of Kernel Compile is shorter comparatively. During the experiments, workloads have been created randomly with above applications, which have been submitted to the cloud pool with a given rate. Each VM will be given a lifetime, when the workload VM runs out of its time, it will be stopped immediately and the resource occupied by the VM will be released.

6.2 Experimental Results

This section summarizes the experimental evaluation of *CloudMon*. Experiment Group 1, 2 and 3 are used to evaluate the suitability of the proposed fuzzy control system for dynamic resource allocation to NIDS virtual appliance with time-varying workloads. The performance comparison of *CloudMon* and Hyperspector [12] are given in Experiment Group 4. Experiment Group 5 is used to test the effectiveness and efficiency of our global resource manager.

Experiment Group1. Before we begin to test the performance of our fuzzy controller, we first evaluate how resource allocation will affect the drop rate of Snort. We set a very low CPU allocation 10% to the *NIDS-VA*, which means that a VM cannot use more than 10% CPU time, even if there are idle CPU cycles. We allocate 192M memory to the *NIDS-VA*. *Tcpreplay* is used to send 100,000 packets at a speed of 50 mbs from a load-generating client.

First, the CPU allocation is first investigated. It is expected that the drop rate could be high, since the CPU allocation is very low. But, unexpectedly, the drop rate of Snort is only 3.5%. We notice that the number of packets that Snort captured is 42,378 which are far less than what the client has sent. At first we thought the drop rate data produced by Snort could be wrong, but through monitoring the *Tx* and *Rx* of *NIDS-VA* reported by *Proc* file system, we noticed that the number of received packets is consistent with the number of the packets Snort has captured. We also observed the number of packets that arrived at the bridge port connected with `Peth0`, and it is almost the same with the number of the packets client has sent. This is to say, some packets have arrived at the bridge, but did not reach the kernel of *NIDS-VA*. We gradually increase the CPU allocation, and the dropped percentage of *NIDS-VA* decreases accordingly. As shown in Table 2, when the CPU allocation reaches 60%, all packets are received by *NIDS-VA*. Therefore, we can come to a conclusion that if the CPU allocation is inadequate, the *NIDS-VA* will also drop packets. A strange phenomenon is that *Snort* has gained relatively more CPU cycles than *NIDS-VA* (*Snort*'s drop rate is relatively low). Generally speaking, the operations in Linux networking system are kernel-mode operations, and they cannot be preempted by user-mode application such as *Snort*. Therefore we thought that the *Snort* process will be starved. This phenomenon is probably related with the scheduling strategy of Xen scheduler and Linux networking subsystem. We also observed that when the CPU allocation of *Snort* is increased the drop rate reported by *Snort* also changes. The actual drop rate consists of two parts: the drop rate *Snort* reports and the drop rate of *NIDS-VA*. In the following experiments, we calculate the drop rate by summing these two parts.

We notice that when the CPU allocation is 100%, *Snort*'s drop rate is 1.5%. Recall that in the above experiment the memory allocation is 192M. We increase the memory allocation to 256M this time and repeat the experiment. The results show that *Snort*'s drop rate decreases, especially for 80% and 100% CPU allocation, *Snort*'s drop rate is nearly 0%.

Table 2. Drop rate for *NIDS-VA* and *Snort* under different CPU allocations

CPU alloc / Dropped by	10%	20%	30%	40%	50%	60%	80%	100%

NIDS-VA	58.6 %	46.8%	20.9%	11.3%	4.4%	0.0%	0.0%	0.0%
Snort	5.5%	5.0%	7.1%	4.1%	3.9%	3.2%	1.7%	1.5%

Experiment Group2. From the first experiment we can see, the performance of *NIDS-VA* can be improved through adjusting the CPU and memory allocation. In this experiment group, we evaluate the effectiveness and performance of our dynamic provision approach for adaptive CPU allocation. To simulate a resource competition situation, a virtual server is running on the same physical machine with the *NIDS-VA* and *CPUburn* is running in it to consume CPU as much as possible. As shown in Figure 13, to simulate time-varying workloads, we change the packets sending speeds every 10 seconds. Figure 14 shows the actual CPU allocation obtained from the fuzzy controller when handling varying network traffic. Three target drop rates (TDR) has been set for the fuzzy controller: 1%, 2% and 3% and we try to figure out the difference of CPU allocations for the three TDRs. First, as we can see from Figure 14, all of them can achieve adaptive CPU allocation to keep up with the time-varying workloads. For 3% TDR, its CPU allocation is smaller than the allocation for 1% and 2% TDRs almost at any time, and it can save about 7% CPU on average compared with 1% TDR.

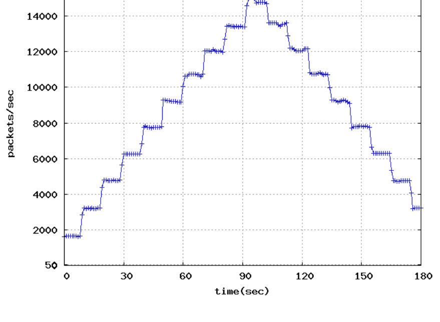
For 1% TDR, the latter part of the curve exhibits more jitters and declines slower compared with 2% and 3% TDR. This is because there exists sudden burst of transient drop rate, which will have a more significant impact for smaller TDR. For example, when encountering sudden burst like 8%, for 1% TDR the deviation is 7%, while for 3% TDR, the deviation is 5%. Therefore, the controller will allocate more CPU for 1% TDR than for 3% TDR. We can also infer that from the following fuzzy rule segments:

IF deviation IS small negative THEN cpu_change IS small negative

IF deviation IS middle negative THEN cpu_change IS middle negative

Figure 15 shows the transient and accumulated drop rate for 2% TDR. We can see that the transient drop rate fluctuates up or down at the TDR, while the accumulated drop rate tends to gradually converge at the TDR. We can also see some transient spikes of drop rate. For example, at the 105th second the drop rate is almost 6%. Most of the transient spikes are abnormal which should be filtered out. A threshold for transient spikes has been set up. Only when the current drop rate exceeds the threshold for two successive observation points, it will be fed back to the fuzzy controller.

Figure 16 shows the variation of accumulated drop rates. For 1%, 2% and 3% TDRs, the accumulated drop rates almost converge at their own TDR respectively. Combined with the results shown in Figure 14, we can see that there is a balance between CPU allocation and the performance of *NIDS-VA*, and our fuzzy controller can dynamically control the CPU provision for *NIDS-VA* to maintain the drop rate at a given target value.



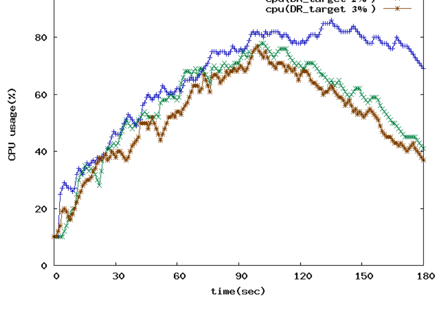


Figure 13: Time-varying workloads

Figure 14: CPU allocation under different drop rate targets

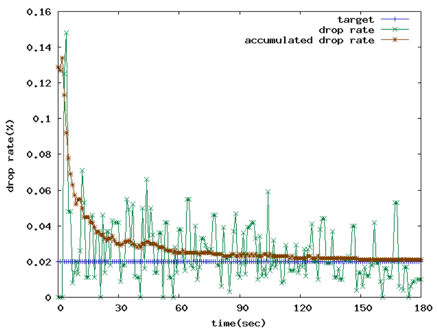


Figure 15: transient and accumulated drop rate for 2% drop rate target

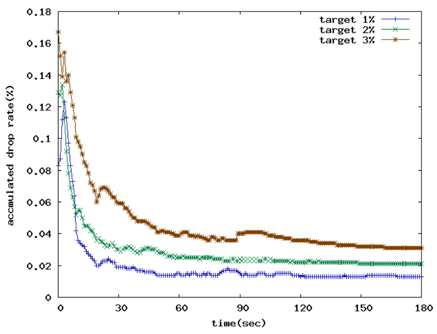
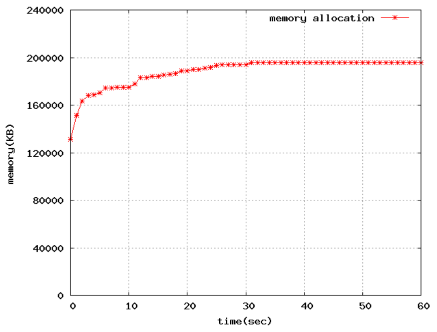


Figure 16: Accumulated drop rate under different drop rate targets

Experiment Group3. To evaluate the effectiveness of our approach in memory adjustment, we first set the initial memory size of NIDS virtual appliance to 128MB, and we observe that the used swap size reaches 69428KB after Snort starts, which indicates that the memory resource is squeezed. We use Tcpdump to generate network traffic at a speed of 200Mbit/s to stress the Snort and the target drop rate is set to 0% - a very stringent requirement. Two methods can be used to adjust VM's memory size in Xen: "xm mem-set dom_id mem_size" in dom0 and "echo –n 'mem_size' > /proc/xen/balloon" in domU. The latter one has been chosen, since it can allocate memory at the granularity of KB. As shown in Figure 18, Snort experiences a severe performance bottleneck at the beginning of the experiment due to the extreme shortage of memory, and its drop rate even reaches 82.6%. Figure 17 shows that the fuzzy controller allocates about 40MB memory in three continuous time intervals and greatly relieves Snort from the performance bottleneck. While at the sixth second, drop rate reaches 20%. This is because the newly allocated memory gets exhausted and the performance of Snort degrades again. The fuzzy controller continuously adjusts the memory allocation to fulfill the performance of Snort based on the drop rate it observed, and after the 31st second, the drop rate maintains around zero. In this experiment, memory allocation is increased at all the time. This is because in Xen 3.2 the memory allocation of a VM cannot be decreased to a value less than 238,592KB. For example, the current memory size is 512MB, and we try to adjust it to 128MB through "xm mem-set", but the actual memory size can only be shrunk to 238,592KB. It also means that once the memory is allocated, it is hard to be reclaimed. To avoid resource over-provision, we modified the fuzzy sets and rules to enable a much finer tuning when the drop rate is relatively low. The experimental results show that memory allocation given by the controller is gradually approaching to an appropriate value based on the observed drop rate.



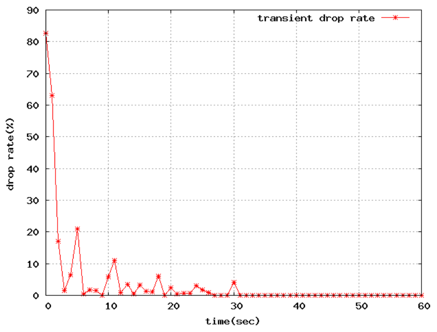


Figure 17. Memory adjustment for

Figure 18. Transient drop rate of

NIDS VA **Snort**

Experiment Group4. Hyperspector [12] is a virtual distributed monitoring framework for intrusion detection. It also adopts a software port mirroring mechanism to enable the *IDS-VA* to capture all the packets that the server VM sends and receives. However, Hyperspector is implemented based on a customized FreeBSD operating system, and the implementations of VMM and software port mirroring mechanism of Hyperspector are significantly different from CloudMon. It is nearly impossible to port them to our Xen-based virtualization platform. Thus we repeat the snort drop rate experiment of [12] in CloudMon, and compare our results with the results of Hyperspector given in [12]. 1-byte UDP packets are sent from an external physical machine to the destination machine which hosts the server VM and *NIDS-VA* at various rates. We also measure the performance of Snort running in base system (Domain 0) of Xen. As shown in Figure 19, CloudMon begins to drop packets when the packet sending rate reaches about 100,000 packets per second, while Hyperspector experienced packet drops at about 50,000 packets per second according to [12]. Generally speaking, CloudMon outperforms Hyperspector to some extent, while CloudMon exhibits more overhead than Hyperspector when compared with base system. In CloudMon, as the network load goes higher, the drop rate of the *IDS-VA* is higher than that in the base system by 0.5% to 1.3%. While in Hyperspector, the performance of IDS-VA is much closer to the base system. Through analysis we found that CloudMon incurs higher overhead compared with the base system in two operations: one is the Linux bridge packet duplication and forwarding; the other is the packet transmission between Xen front-end and back-end drivers.

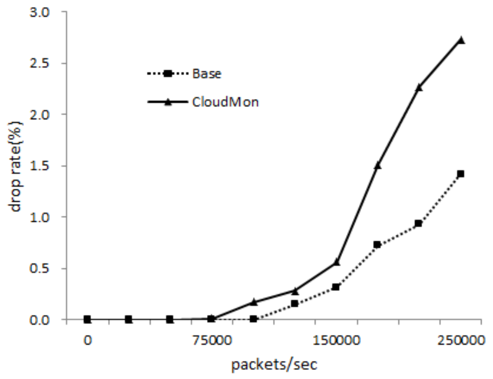


Figure 19. Packet drop rate of Snort

Experiment Group5. The purpose of this experiment group is to evaluate the effectiveness and the efficiency of our Cloud-Oriented resource management approach. Before enabling our algorithms, our cloud pool has been running a series of VMs. To better simulate the behaviors of a dynamic cloud computing environment with VM arrival and departure, we submit new application VMs to the cloud pool according to a Poisson distribution during the running of our online algorithm. To evaluate the performance of our algorithm, we compared it with *Static First Fit (SFF)* [3], *and Static Best Fit (SBF)* in terms of the number of online physical hosts and the pool resource utilization. Correspondingly, our online algorithm is named *DBF*. Figure 20 shows that our algorithm uses less number of online physical hosts to run the application VMs than *SFF* and *SBF* algorithm. At the 245th minute, our algorithm only uses 28 physical hosts, saving 10 and 6 nodes compared with *SFF* and *SBF,* respectively. The reason is that our algorithm exploits live migration to ensure VMs and *NIDS-VAs* are concentrated at all the time, thus achieve the objective of resource efficiency. Figure 21 shows the overhead incurred by distributed NIDS system. The resource utilization is an average of two resource dimensions: CPU and memory. As

[3] Static means no migrations of VMs.

shown in Figure 21, an average of 15% overhead is incurred by the distributed NIDS system，which means the distributed NIDS system additionally consumes about 1/6 resources compared with the overall resource consumption of application VMs. During the experiments, we found that the network traffic of a VM may be monitored twice, since the communication between two VMs hosted on different hosts may be inspected by both the *NIDS-VA*s of the two hosts. It also means that if we carefully manage the traffic being inspected and avoid duplicated monitoring, the overhead can still be reduced. The overhead is below 20% in the worse case, which is acceptable in a cloud environment with high security requirement.

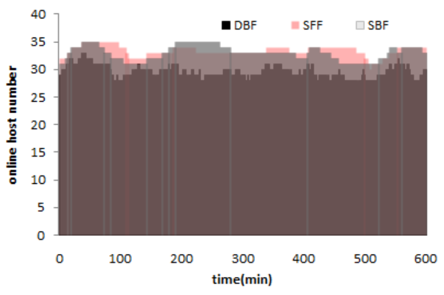


Figure 20: Online host numbers Comparison of three algorithms

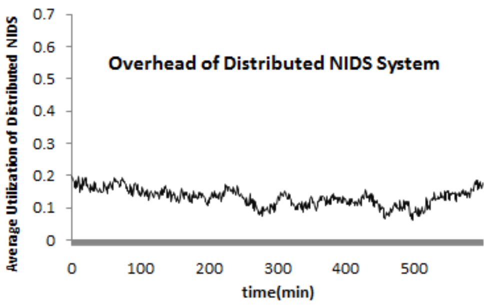


Figure 21: Average utilization of distributed NIDS system

7. RELATED WORK

Cloud computing has been recognized as the enabling technology of IoT. Some cloud systems and services have been successfully designed and developed to serve as the underlying infrastructures for storing and handling IoT data and applications. IoTCloud [3] is a cloud-compatible open source messaging system and extendable API which enables developers to write scalable high performance IoT and sensor-centric applications. Sensor-Cloud [4] uses SensorML, a sensor model language, to describe sensor metadata and manages sensors via the cloud, rather than providing their data as a service. SenaaS [5] proposes a sensor-as-a-service model and encapsulates both physical and virtual sensors into services which can provide an event- driven sensor virtualization approach for Internet of Things Cloud. Gubbi etc. The study [2] presents a Cloud centric vision for worldwide implementation of Internet of Things, and implements a cloud-based IoT application using Aneka cloud service and Microsoft Azure.

One challenge which may pose negative impact on the application of cloud technology in IoT is security. To monitor the security of cloud environments, virtual security appliance has been proposed and shown great market potential in cloud security markets. A recent report [9] from IDC pointed that "virtual security appliance deliver a strong left hook to traditional security appliances" and "radically change the appliance market". IDS and IPS being parts of traditional security appliances are now virtualized as virtual appliances and produced by many network security vendors.

7.1 Virtualized Networked Intrusion Detection Systems

Virtualization technology has long been used by security applications to enhance the security of computer systems [10][11][12][24]. The basic idea is to use VM to isolate these security applications from the protected systems, and leverage VMM to monitor and protect them without having to trust the operating system. In academia, researchers have adopted virtual machine technology to enhance the intrusion detection systems. Livewire [10] leverages virtual machine technology to isolate the IDS from the monitored host, while it still can enable IDS VM to monitor the internal state of the host through the VM introspection technique. Joshi et al. used vulnerability-specific predicates to detect past and present intrusions [11]. When vulnerability is discovered, predicates are determined and used as signatures to detect future attacks. HyperSpector [12] is a virtual distributed monitoring environment used for intrusion detection, which provides three inter-VM monitoring mechanisms to enable IDS VM to monitor the server VM. In most of the above systems, IDS VM shares the physical resource with the host and the other VMs on the same machine. Sharing will bring resource contention and impact the performance of IDS, but neither of them has considered the performance issues.

7.2 NIDS Performance Modeling and Resource Provision

Many research studies focus on the performance issues of NIDS. Several proposed NIDS systems have been tested in respect of their performance [13][14]. While these approaches are only used for performance analysis and evaluation, neither of them considered the relationship between performance and resource usage. Lee et al. [15][16] proposes dynamic adaptation approaches which can change the NIDS's configuration according to the current workloads. Dreger et al. [17] proposed a NIDS resource model to capture the relationship between resource usage and network traffic. They use this model to predict the resource demands of different configurations of NIDS. Both of them focus on NIDS configuration adaptation, while the implementation of adaption capability depends on the implementation details of NIDS to some extent, the mechanism implemented in one NIDS may not be fit for others. By contrast, our approach leverages a feedback fuzzy control mechanism to dynamically provision resources to NIDS application to fulfill its performance requirements without the need to give a model to estimate its resource usage. Xu et al. [18] presented a two-layered approach to manage resource allocation to virtual containers sharing a server pool in a data center. The local controller also uses fuzzy logic, while in that paper fuzzy logic is used to learn the behavior of the virtual container instead of online feedback control. Besides, the proposed method is essentially concerned with server applications, not for NIDS. Different from the above systems, our system leverages feedback fuzzy control mechanism to achieve adaptive resource provision for NIDS.

7.3 Resource Management in Distributed Systems

Dynamic resource allocation and management in distributed systems has been studied extensively, early research experiences [25][27][32] focus much on how to schedule independent or loosely-coupled tasks in a shared system. The objective is to balance the workload among servers, so as to maximize system throughput. In [32], resource provisioning for large clusters hosting multiple services was modeled as a bidding process. However, these existing techniques are not directly applicable for the resource provision of virtual machine. They lack of good isolation mechanisms, thus it is hard to meet the QoS requirements of applications in VMs. VM Migration has been used to by [33] to handle overloaded physical nodes. The black box strategy uses utilization statistics to infer which VMs need to be migrated, and uses application statistics to infer the resource requirements using a queuing theory. However, it is very difficult to predict the resource requirements of complex applications requiring multiple resources using queuing theory. PMapper [26] uses live migration technique to enable resource scheduling and design migration strategies to guide the application placement for energy saving purpose. While none of the above systems considered the resource provision problem for network monitoring and the distributed NIDS deployment.

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed *CloudMon* which is a dynamic resource provision and management system for distributed *NIDS-VAs* deployment and monitoring in a cloud environment. We use fuzzy control method to characterize the complex relationship between performance and resource demands to overcome the absence of mathematical model for NIDS virtual appliance. An online fuzzy controller has been developed to adaptively control the resource allocation for NIDS under varying network traffic based on a set of linguistic rules. We identified the problem as multi-dimensional vector packing problem, and adopted an offline VM placement strategy to improve resource efficiency. An event-driven online VM mapping algorithm is proposed to deal with the dynamics of a cloud environment. We have implemented our approach in the Xen-based iVIC platform and our experiment results show that it is a viable solution.

Our ongoing work is three-fold. First, we plan to take more practical evaluation through real applications on our iVIC platform to further investigate the effectiveness and efficiency of *CloudMon*. Second, we will extend *CloudMon* to support more virtualization platforms such as Xen, VMware. Third and last, we will collect more real-life network traffic to train our fuzzy controller and improve its precision.

ACKNOWLEDGMENT

The authors gratefully acknowledge the anonymous reviewers for their helpful suggestions and insightful comments to improve the quality of the paper. The work reported in this paper has been partially supported by National Nature Science Foundation of China (No. 61202424, 61272165, 91118008), China 863 program (No. 2011AA01A202), Natural Science Foundation of Jiangsu Province of China (BK20130528) and China 973 Fundamental R&D Program (2011CB302600).

REFERENCES

[1] L. Atzori, A. Iera, G. Morabito, The Internet of Things: A survey, Computer Networks 54 (2010) 2787–2805.

[2] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, Marimuthu Palaniswami. Internet of Things (IoT): A Vision, Architectural Elements, and Future Directions. Technical Report, Submitted on 1 Jul 2012.

[3] G.C. Fox, S. Kamburugamuve, R.D. Hartman, Architecture and Measured Characteristics of a Cloud Based Internet of Things API, International Conference on Collaboration Technologies and Systems (CTS), 2012.

[4] M. Yuriyama and T. Kushida, Sensor-Cloud Infrastructure - Physical Sensor Management with Virtualized Sensors on Cloud Computing, 13th International Conference on Network-Based Information Systems (NBiS), Sept. 2010, pp. 1-8.

[5] Alam, S.; Chowdhury, MMR; Noll, J.; SenaaS: An Event- driven Sensor Virtualization Approach for Internet of Things Cloud, IEEE International Conference on Networked Embedded Systems for Enterprise Applications (NESEA), 2010

[6] Virtual security appliance. Http://en.wikipedia.org/wiki/Virtual_security_appliance

[7] M. Roesch, Snort - Lightweight Intrusion Detection for Networks, in Proceedings of the USENIX LISA '99 Conference, November 1999.

[8] Markatos E, Antonatos S, Polychronakis M and Anagnostakis K, Exclusion-based signature matching for intrusion detection, in Proceedings of the IASTED international conference on communications and computer networks, CCN, 2002.

[9] Virtual Security Appliance Survey: What's Really Going On? http://www.idc.com/getdoc.jsp?containerId=220767

[10] T. Garfinkel and M. Rosenblum, A Virtual Machine Introspection Based Architecture for Intrusion Detection, in Proceedings of the 10th Annual Network and Distributed System Security Symposium, Feb. 2003.

[11] A. Joshi, S. T. King, G. W. Dunlap and P. M. Chen, Detecting Past and Present Intrusions through Vulnerability-specific Predicates, in Proceedings of the 2005 SOSP, Oct. 2005.

[12] K. Kourai and S. Chiba, Hyperspector: Virtual distributed monitoring environments for secure intrusion detection, in Proceedings of the 1st ACM/USENIX International Conference on Virtual Execution Environments, 2005.

[13] V. Paxson, Bro: A System for Detecting Network Intruders in Real-Time, Computer Networks, vol. 31, no. 23-24, 1999, pp. 2435-2463.

[14] C. Kruegel, F. Valeur, G. Vigna and R. Kemmerer, Stateful Intrusion Detection for High-Speed Networks, in Proceedings of IEEE Symposium Security and Privacy, IEEE Computer Society Press, Calif., 2002.

[15] W. Lee, J. B. Cabrera, A. Thomas, N. Balwalli, S. Saluja, and Y. Zhang, Performance Adaptation in Real-Time Intrusion Detection Systems, in Recent Advances in Intrusion Detection, 2002.

[16] W. Lee, W. Fan, M. Miller, S. J. Stolfo, and E. Zadok, Toward Cost-sensitive Modeling for Intrusion Detection and Response, Journal of Computer Security, 10(1-2):5–22, 2002.

[17] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer, Predicting the resource consumption of network intrusion detection systems, in Recent Advances in Intrusion Detection, R. Lippmann, E. Kirda, and A. Trachtenberg, Eds., 2008, pp. 135–154.

[18] Xu, J., Zhao, M., Fortes, J., Carpenter, R., Yousif, M.: Autonomic resource management in virtualized data centers using fuzzy logic-based approaches. Cluster Comput. J. 11, 213–227, 2008.

[19] Jan Jantzen, Foundations of Fuzzy Control, John Wiley & Sons, 2007.

[20] M. R. Garey, R. L. Graham, D. S. Johnson, Andrew. M. R. Garey, R. L. Graham, D. S. Johnson, Andrew. Journal of Combinatorial Theory. Journal of Combinatorial Theory, Vol. 21 (1976), pp. 257-298

[21] Miller, R. E., Thatcher, J. W. , "Reducibility Among Combinatorial Problems", In Complexity of Computer Computations, pp. 85-103, New York, 1972.

[22] G. Galambos, G.J. Woeginger, On-line bin packing: a restricted survey, Mathematical Methods of Operations Research 42 (1) (1995) 25-45.

[23] BitTorrent. Http://www/bittorrent.com.

[24] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A. D. Keromytis Detecting Targeted Attacks Using Shadow Honeypots. Proc. of the 14th USENIX Security Symposium, 2005.

[25] Fu, S. and Xu, C. 2004. Migration Decision for Hybrid Mobility in Reconfigurable Distributed Virtual Machines. In Proceedings of the 2004 international Conference on Parallel Processing (August 15 - 18, 2004). ICPP. IEEE Computer Society, Washington, DC, 335-342.

[26] Akshat Verma, Puneet Ahuja, Anindya Neogi, "pMapper: Power and Migration Cost Aware Application Placement in Virtualized Systems", Middleware 2008: 243-264.

[27] Shen, K., Tang, H., Yang, T., and Chu, L. Integrated resource management for cluster-based internet services. ACM SIGOPS Operating Systems Review 36, SI (2002), 225-238.

[28] Chetan S Rao, Jeffrey John Geevarghese, Karthik Rajan. Improved approximation bounds for Vector Bin Packing. Technical Report.

[29] Wang, Z., Zhu, X., Singhal, S., "Utilization and SLO-based control for dynamic sizing of resource partitions", In Proceeding of 16th IFIP/IEEE Distributed Systems: Operations and Management (DSOM), October 2005.

[30] Bo Li, Jianxin Li, Jinpeng Huai, Tianyu Wo, Qin Li, and Liang Zhong. Enacloud: An energy-saving application live placement approach for cloud computing environments. In Proceedings of the 2009 IEEE International Conference on Cloud Computing, pages 17-24, Bangalore, India, 2009.

[31] SPEC CPU2006, next-generation, industry-standardized, CPU-intensive benchmark suite. http://www.spec.org/cpu2006/.

[32] Chase, J., Anderson, D., Thakar, P., Vahdat, A., and Doyle, R. Managing energy and server resources in hosting centers. In Proc. of Symposium on Operating Systems Principles (SOSP) (October 2001).

[33] Wood, T., Shenoy, P. J., Venkataramani, A. and Yousif, M. S. Black-box and gray-box strategies for virtual machine migration. In NSDI (2007), USENIX.