Distributed Policy Processing in Active-Service based Infrastructures

Wei Yu, Sriram Chellappan[+], Dong Xuan[+], and Wei Zhao

Computer Science Department,

Texas A&M University, College Station, TX 77843

Email: {weiyu,zhao}@cs.tamu.edu

[+]*Department of Computer and Information Science,*

The Ohio-State University, Columbus, OH 43210

Email: {chellapp,xuan}@cse.ohio-state.edu

Abstract

More and more applications in the Internet are requiring an intelligent service infrastructure to provide customized services. In this paper, we present an infrastructure, which can transparently and effectively provide customized active-services to end users and dynamically adapt to changing customized policies in large distributed heterogeneous environments. The infrastructure consists of two components: the policy agent and middleware box. Particularly, our technologies include: 1) Generic active-service based infrastructure, where the policy agent can integrate policies requested by applications, and middleware boxes can transparently execute services and 2) Distributed policy processing in the middleware box. We study two policy partitioning schemes to achieve conflict-free policies for distributed policy processing and guarantee the correctness of the policy execution. We conduct extensive performance evaluations on different schemes proposed. Our experimental results demonstrate that our policy partitioning schemes can effectively generate partition-capable and conflict-free policy sets. The evaluation results also show that distributed policy processing can achieve over 70 percent increase in performance/price ratio with proper assignment of the policy distribution degree compared to a purely centralized approach.

Keywords: *Policy processing, Active Networks.*

I. Introduction

In this paper, we present a distributed policy processing infrastructure which can effectively deploy customized services. We also address the effectiveness of policy processing issues in this infrastructure. Our proposed distributed policy processing schemes can effectively support large number of policies requested by increasing number of applications and services.

With the number of new applications and end users increasing, services to be supported in the Internet are increasing. For example, services like VPN (virtual private network), NAT (network address translator), access control, layer 4/6 routing, and content delivery services are becoming basic services, and newer services keep coming up in the Internet. At the same time, customized policies requested by services are becoming more complex. For example, some services such as content delivery require checking some customized information in the packet payload [1] [2]. Services like dynamic VPN service, dynamic-coalition service require the service to be deployed for a limited time and are either automatically torn down or canceled by the user. These services require the system to be endowed with the capability to dynamically modify the network's application supported profile and provide on-demand services. Some emerging services such as active worm defenses have become very important due to several such attacks in the recent past. Coupled with their ability to propagate fast, worms have caused significant economic losses [3].

Motivated by above observations, we need a network infrastructure provide the 'active service', which means that we have to efficiently integrate existing services transparently, quickly deliver new services, and intelligently adapt to changing network environments. By leveraging the existing active network results (breaking with tradition by allowing the network to perform customized computation on the user data [4] [5]), we study an active-service based infrastructure in this work. In our proposed infrastructure, two components are involved: the service policy agent and middleware boxes. The service policy agent, as the service request interface can effectively integrate the application policies and distribute the corresponding policies to the middleware boxes. The middleware boxes deployed in some functional network locations construct an overlay network and cooperate in a distributed fashion to upwardly provide services required by applications and downwardly adapt to heterogeneous networks. In this infrastructure, service creation is very simple from the user's perspective. The user/ application service creators only need to submit their service requirements to the service policy agent. The service requirements can also be changed during run-time by the end users. The service can be automatically created/ updated by the cooperation of service policy agent and middleware boxes, i.e., the service policy agent translates and integrates the new service policies and/ or updates service policies and distributes it in an understandable format to corresponding middleware boxes. The middleware boxes execute the policies to achieve customized service execution dynamically during system run-time. The middleware box is the basic functional entity in the infrastructure for policy execution. It plays an important role in the efficiency of policy processing to support increasing number of services with complex policy requirements. In this paper, we conduct an extensive study on policy processing in the middleware box. Specifically, the contributions of our paper are:

- We propose a generic active-service based infrastructure. We integrate the policies requested by applications and middleware solutions to transparently execute services in heterogeneous networks. The middleware are equipped with policies that are generated by the policy agent, which translates various application service requirements.
- We propose distributed policy processing schemes in the middleware box. By distributed policy processing, we mean that the line cards of middleware boxes can deploy a portion of policy rules and make the policy matching and processing efficient. We study two policy partitioning schemes to achieve conflict-free policies in the case of distributed policy processing and guarantee the correctness of the policy execution. We then discuss enhancements to our approach using available statistical information of policy rules to increase the effectiveness of the policy partition.
- We conduct extensive performance evaluations on different infrastructures and schemes. The evaluation results show that our distributed policy processing can achieve over 70 percent increase in performance/price ratio with proper assignment of the policy distribution degree, compared with a centralized approach. The experimental results also demonstrate that in order to make the policy both partition-capable and conflict-free, newer policies have to be generated, which can be effectively reduced by using policy compression schemes.

The rest of paper is organized as follows: In Section II, we discuss necessary background and related work in this area. In Section III, we present an active-service based infrastructure with two important components: service policy agent and middleware box. In Section IV, we present the distributed policy processing in the middleware box and describe policy partition-capable algorithms. In Section V, the simulation and evaluation results are presented. The summary of this paper and future work are given in Section VI.

II. Background and Related Work

In this section, we discuss background and existing work related to providing ‘active service’. These include work on active networks, policy processing/ packet classification, and policy-based routing.

Traditional data networks passively transport bits from one end system to another. Due to the Internet end-to-end design philosophy, the network just conducts the packet forwarding and its role in the network is limited [4]. Active networks actually allow the user to inject customized programs or deploy policies into the network during run-time, where the routers or switches in the network will perform customized computations on the messages passing through them. For example, a user in the active network could send a “smart packet” to each router and arrange for the program to be executed when their packets are processed [5]. The fundamental benefit of the active network is to provide a powerful way for the user or application to drive the customization of network infrastructure and allow new services (active services) to be deployed at a faster pace than possible in traditional network infrastructures. Much work has been done in active network research area. SPIN [6] and SwitchWare [7] are such examples. Most of current work on the active networks focuses on designing a platform within the single node to execute the code-embedded packet [8] [9]. Some work also studied issues in providing active services by designing application-specific support services such as, video conference transcoding proxy [10], content-aware gateway [11], NAT [12], customized many to one communication [13], customized multicast communication [14] etc. In this paper we mainly focus on efficiently integrating services, transparently providing services and intelligently adapting to changing network environments for applications in a distributed fashion.

Depending on the service requirement, policy execution can have impacts on the network packet, as the service policies are normally executed in the network core device – Routers and Switches. The policy execution needs to match the run-time packet with the deployed policies. This policy match problem can be generalized by the packet classification problem defined as follows*: A policy table has N rules –* R_j *:* $<C_i => A_i>$*, where* $1 \leq j \leq N$ *and* R_j *contains two parts: 1) Condition* C_i*:* $C_i[1], C_i[2], C_i[3], \ldots C_i[D]$*, a D-tuple, where D is the policy dimension; 2) Action set* A_j *with j actions from the system action set with M actions* $(1 \leq j \leq M)$*. For an incoming packet P with the header considered as a t-tuple* $(P_1, P_2, \ldots, P_t)$*, where* $1 \leq t \leq D$*, the packet classification problem is to find the m rules* R_m $(1 \leq m \leq N)$ *matching among N rules with D-tuple, such that* P_t *matches* $C_m[t]$*,* $\forall 1 \leq t \leq D$. We call these R_m rules as the ‘best’ matching set for the incoming packet *P*. For example, *R = <(1010*, *, TCP, 1024-1080, *), DENY>* is policy rule with *condition* C_1*=(1010*, *, TCP, 1024-1080)* and *action* with *DENY*. Then, the packet with header *(10101….1111, 1110..000, TCP, 1050, 3)* matches the *Condition* C_1, and is therefore dropped. The packet with header *(111111…000, 1111..001, UDP, 1024)* does not match the condition C_1 and is therefore forwarded. Computational Geometry theory has been shown that in its fully generality, packet classification requires either $O(logN^D\text{-}1)$ time and linear space or *logN* time and $O(N^D)$ space, where *N* is the number of rules and *D* is the number of header fields used for matching. Specifically, the IP routing lookup is one of the simplest examples of the above problem, where *D* is 1 and the action is always forwarding. Much work has been done on effective IP routing lookup schemes such as binary tries, LC tries and controlled prefix expansion [15]. Due to service requirement such as network traffic management, firewalls, VPN, high dimensional packet classification/ policy processing has been studied with interest and approaches like [16] can effectively work at two-dimensional packet classifications. Bit Parallelism and RFC (Recursive Flow Classification) [17] and Cisco Turbo ACL [18] introduce new packet classification schemes which only work well in average cases. Work in [19] translates the high dimension packet classification problem to low dimension assuming that the utilized space in some dimensions are small; TCAM (Ternary Content-Addressable Memory) [20] provides hardware-based algorithms to support parallel matching for different fields, which currently are only suitable for small policy tables. Generally, most of the solutions are service specific or have some limitations and are not suitable for policy processing in our infrastructure. Our distributed policy processing is more generic and can effectively provide different service requirements.

Due to requirements on QoS (quality of service), policy-based routing defines a router/switch as one, configured to use different criteria than just a distance metric to decide which peer to forward a packet to. Policy routing is defined to configure a router to inspect and modify the attributes of routes to provide a flexible mechanism to route IP traffic to a destination with QoS requirement. Especially, the policy-based routing is one type of 'active service' for QoS, which is actively reconfiguring the packet path to satisfy the QoS requirement. In [21], the policy routing problem is initially defined with three models: policy-based distribution of routing information, policy-based packet filtering/ forwarding, and policy-based dynamic allocation of network resources. Several products have implemented this feature [22] [23]. Most of policy-based routing schemes only focus on the IP layer to provide QoS. Our approach targets policies in different layers. Thus, our approach extends work on policy-based routing and provides a generic platform to support requirements of different services.

III. Active-Service based Infrastructure

In this section, we present the active-service based infrastructure. We first give an overview of the proposed infrastructure and then introduce its basic components and its service workflow.

A. Overview

Service deployment is the fundamental aspect of the promise of next-generation network applications. With unprecedented demand to create and deploy new revenue-generating and cost saving services quickly, the next-generation networks are compelled to provide scalable service deployment infrastructures to enable the service creation. Active networks break with tradition by allowing the network to perform customized computations on the user data. For example, a user on top of an active network could send a customized compression program to a node in the network and request that the node execute the program when processing its packets. However, allowing the user to directly configure the network brings several management and security concerns to the network. We leverage the existing active network results and present an active service based infrastructure, which can be used to efficiently integrate services, transparently provide services, and intelligently adapt to changing network environments for applications.

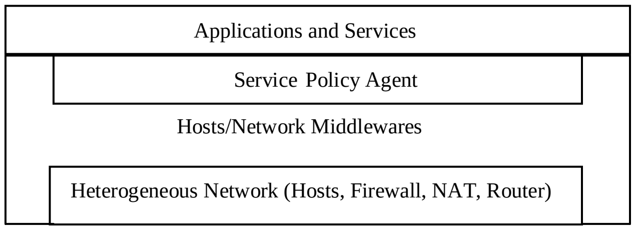
Applications and Services
Service Policy Agent
Hosts/Network Middlewares
Heterogeneous Network (Hosts, Firewall, NAT, Router)

Figure 1: Host/Network Middleware

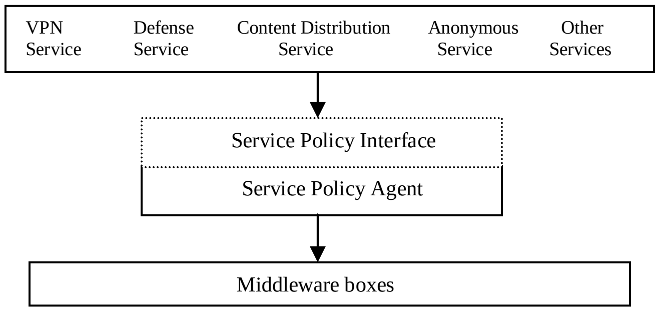

Figure 2: Policy Agent-based System

In this infrastructure, there are two important components: network service policy agent and middleware-box. In simple terms, the network server policy agent becomes the interface between the user/ application and network itself. It can effectively integrate the user/ application requirements and translate them to a format understood by the programmable network devices, i.e., middleware boxes. The middleware boxes execute the customized policies to achieve the service deployment in the network. The middleware boxes can be deployed in some functional locations (host, network edge or network core) as a middleware in the network. They construct an overlay in the network data plane and cooperate in a distributed fashion to upwardly provide and activate services required by applications and downwardly adapt to heterogeneous networks as shown in Figure 1.

B. Components

1) Policy Agent

The main functionality of the service policy agent is policy translation and global consolidation in the system. The functional components of the service policy agent are shown in Figure 2. All users/ applications provide their customized service requirements such as, VPN traffic requirement matrix, firewall access control list, intrusion detection requests etc to the service policy agent. The service policy agent can now translate the high-level application requirements into the defined policy format for the middleware box. With secured policy communication protocols, the middleware box can obtain and execute the policies for customized services. According to application/ service requirements and the network information, the service policy agent includes the following functionalities:

The correction check of application/service policy requirements: It includes two separate tasks. The first one is the syntax check, which checks the syntax of the application policies. The other one is the consistency check, which checks for any conflicts between different policies [24].

Static resource optimization: This task has the objective of maximally satisfying the service requirements with the constraints of system resources. For example, VPN users may have multiple aggregation bandwidth requirements from one network to another; our service policy agent can find optimal routes to globally satisfy the application requirements with the limited network bandwidth constraints by using previous results [25] [26].

Policy optimization: Here, middle-level policy code for each middleware box will be generated and redundant functions among services are deleted. Therefore, the execution policy can be optimized to a certain degree. As our infrastructure supports dynamic service creation, policies for executing services can be changed at system run-time.

Policy generation and distribution: According to the network information and service requirement, the service policy agent can find the service logic and locations to deploy policies. Then, it will distribute the policy to corresponding middleware boxes.

2) Middleware box

Middleware normally refers to the software layer between the operating systems – including the basic communication protocols and the distributed applications that interact via the network [27]. Our middleware box can be software/ hardware inserted between physical networks and applications that transparently and efficiently maps application requirements to what the network provides, similar to the functionality of operating systems. In this sense, we term our programmable device as the middleware box. The middleware box is the basic unit to integrate and execute policies in the infrastructure. Figure 3 shows the component architecture of the single middleware box with following modules:

Signaling Module (SM): This module provides the communication mechanism between the middleware box, service policy agent, and other network entities to exchange the network information and policy rules.

Policy Parse Module (PPM): This module provides the basic parser functionality to support the policy configuration of the middleware box which is similar to the Cisco IOS command parser in router/ switch platforms. This module takes the output from the service policy agent.

Policy Integration Module (PIM): This module takes the input from different modules, i.e., run-time system data, policy parameters and effectively integrates policies. This module also takes the responsibility for the policy partition, which will be described in a later section. The output of this module will be downloaded to the policy rule table in the data subsystem.

Traffic Processing Module (TPM): This module is related to the data subsystem, which is responsible for processing the on-going traffic according to policy rules. The status variables maintain some run-time system information from the traffic processing module and provide this information to policy integration module.

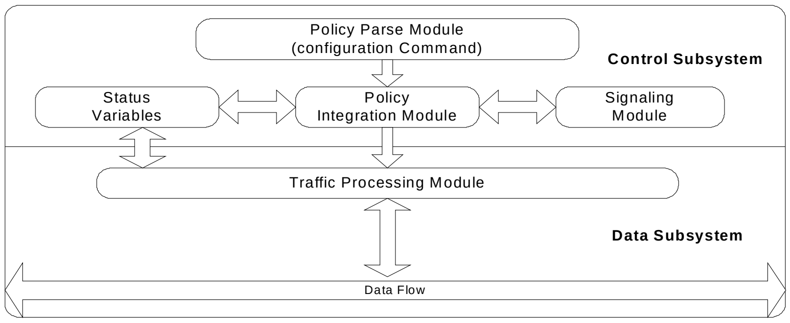

Figure 3: Middleware Box Architecture

C. Workflow

Creating new services in distributed networks require a unique approach. In this section, we introduce the service creation procedure and demonstrate some service examples which can be deployed in our infrastructure. The service is defined as network capability or capability made available to end users and applications. In order to deliver a service, the service logic must be created, distributed to the network and executed by the network equipments. The detail workflow is given below.

Service creation: The service logic is actually created. The user/ application service creator can submit its requirement, which can be processed by service policy agent. Based on the system information, the service policy agent translates, analyzes and generates the corresponding policy rules for corresponding middleware boxes.

Service logic distribution: The translated policies are distributed to the place where it is needed to be applied. We assume the service policy agent is domain-based device and only controls middleware boxes in its own domain. The following generic communication protocols between the service policy agent and middleware-box can be applied: 1) Secured topology auto-discovery: When the middleware boxes and service policy agent are deployed in the network, the service policy agent broadcasts advertisement messages encrypted by its private key to advertise itself as the policy controller with limited TTL value. Only middleware boxes located within a certain network area can receive the advertisement message. When the middleware box receives this message, it will decrypt the message with its configured agent's public key to authenticate the identity of the agent. If the authentication procedure is successful, the middleware box will forward the advertisement to its neighbors. Finally, the middleware boxes and service policy agents will build a self-organized virtual overlay network to provide services. 2) Secured policy download: After the middleware box finds its service policy agent, it will send a request to the service policy agent and get the corresponding policies. Care must be taken about the integrity verification during downloading of the policies. 3) Run-time control protocol: During the system run-time, the service policy agent updates new service requests to the middleware box according to the new service requirements or the policy changes for the existing services. 4) Inter-domain policy agent protocol: When networks become large, hierarchical domain-based approaches will make our system scalable. In order to coordinate the policy execution in multiple domain environments, domain policy agents in different domains are grouped as an inter-domain controlling system to coordinate the policy execution globally.

Service execution: When the policies are received by a middleware box, it will translate them to a format that it can locally understand, such as a policy table. With the policy execution of middleware box, the network is actually changed to deliver the service. This is typically in the middleware box that acts in real-time on packets carried by the network.

According to above workflow, we demonstrate an example taking the case of active worm defense services. Active worms, exploiting security or policy flaws in the system can spread throughout the Internet in a very short time [3]. As the middleware box is a device that is inserted across various locations in the network, it is external to the network. With this strategy, no change is needed to the current network and protocols and highly transparent worm defense services can be achieved. The defense system consists of the following. 1) The service policy agent autonomously translates the defense requirement and selects a number of middleware boxes as the worm detection overlay system. The worm detection system can provide the service to detect the presence of worm attack in the network by monitoring worm scan traffic. We assume that worm detection system can effectively detect the worm by using monitoring methods [28]. 2) The service policy agent connects to the corresponding middleware boxes and deploys the worm monitoring policies. For example, the middleware boxes monitor the worm scanning traffic, i.e., the scanning traffic to unused IP addresses and inactive port (inline with existing work like Honeypots and Internet motion sensor [29][30]). When one middleware box detects potential worm intrusion events, it will send alert events to the service policy agent. As the service policy agent has more network information, i.e., network topology, attack alert events from other network middleware boxes, it can make an worm detection and defense decision, i.e., finding the location of the worm attacker, allocating system limited defense resource near to the worm attacker to perform the worm traffic throttling, letting other middleware boxes in its domain cooperate etc. In this sense, the worm detection and prevention can be effectively and quickly achieved by the cooperation of network-based middleware. 3) The service policy server deploys the worm defense policies to corresponding middleware boxes, like blocking certain active attack flows and blocking traffic from certain worm infected networks.

IV. Policy Processing in Middleware box

The middleware box is the basic functional entity in the architecture for policy execution. It plays the key role in the efficiency of policy processing to support increasing number of services with complex policy requirements. In this section, we will first present the distributed policy processing model and then study the policy partition problem which makes the distributed policy process feasible.

A. A Distributed Policy Processing Model

1) Processing Models

We take the generic architecture of the IP router as the base line design approach for the middleware box, which includes the supervisor card, the router backplane and line cards [31] [32]. The most important module for the policy processing is *PPE* (Policy Processing Engine), which can be developed by expensive *TCAM* [22]. The *PPE* can be located at the supervisor and line cards. The *PPE* maintains the policy rule tables, performs the policy matching and executes the corresponding policies for the incoming packets. In the following context, we will use *LPPE* to represent the *PPE* located at the line card and *SPPE* to represent the *PPE* located at the supervisor card. We assume that the *PPE* is the most expensive subsystem in the middleware box and it is the key module for the performance of the middleware box.

Depending on the location of the *PPE*, we have the following two models: 1) *Centralized Policy Processing*: The *PPE* is located only at the supervisor card. While this type of architecture has been widely deployed in current designs, policy processing cannot be sped up due to centralization being a bottleneck. 2) *Distributed Policy Processing*: Each inbound/ outbound line card maintains the *LPPE*. The policy matching procedure is executed for an incoming packet at the *LPPE* located at the line card locally. If a match cannot be found, the packet will be forwarded to the supervisor card, which conducts the policy matching and execution through the *SPPE*. We emphasize here that for a particular policy, the delay in finding a match is less if it is found in the *LPPE* than when it has to go the *SPPE*.

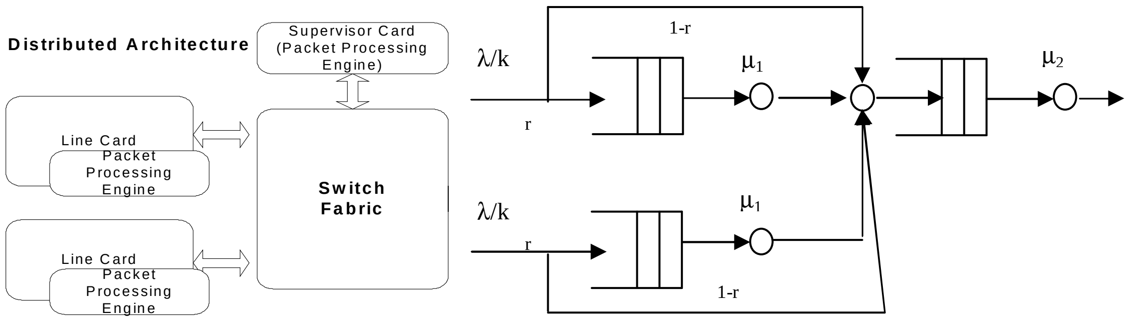

Figure 4: Distributed Policy Processing and Queuing Model

The Distributed Policy Processing approach gives us additional flexibility in policy distribution among *LPPE* and *SPPE*, hence achieving high scalability and efficiency. We describe it in detail below.

In the distributed policy processing model, portion of policies can be deployed at the *LPPE* locally. One fundamental question is how to select subset of policies to be deployed at the line cards. The most intuitively way is just random selection, where each policy has equal chances to be selected. However, an alternate, albeit better approach to use the statistical information of policy rules during deployment of policies that can effectively increase the policy matching hit rate at *LPPE* and improve the overall packet policy matching delay. The detailed scheme is described in the following.

In the distributed policy processing model, suppose the middleware box has deployed *N* number of policies, the *Policy Distribution Degree* defines the percentage of *N* policies to be deployed at the *LPPE* in the line cards locally. In this paper we let the notations *r* and *PDD* to denote the *Policy distribution Degree* interchangeably. When *PDD* ($\in [0,1]$) is 0, the policy processing becomes purely centralized. Figure 4 shows the distributed policy processing and its corresponding queuing model [33]. As there are C_N^{rN} possible choices, one question is how to find the best one. Practical data from real routers show that the routing table at higher speed backbone routers contain few entries with prefixes longer than 24-bits. This is verified by a plot of prefix length distribution of backbone routing table from real configuration devices. For example, 99.93% of the prefixes are 24 bits or less [17]. The distributed policy processing can be enhanced by making use of this fact.

We can assign each policy R_i: $<C_i, A_i>$ with a weight W_i in the policy table with *N* entries. The weight W_i can be calculated according to the probability information of the condition C_i or other considerations. After sorting the partitioned policy rule table based on the weight W_i assignment, we get *r*N* number of policies with largest weight assignment and deploy them in the line cards. This improves the probability of finding a matching policy in the line card locally improving the performance. As the C_i is composed of *k-tuple* $<C_{i1}, C_{i2}, \ldots, C_{ik}>$, the field C_{ij} in the policy table is assigned weight W_{ij} according to the known probability distribution information. As we assume that the probability of C_{ij} for all $j = 1, \ldots, k$ are independent, the weight for R_i can be easily calculated by $\prod_{j=1}^{k} W_{ij}$. For the policy distribution schemes, we have: 1) Weight assignment *(WA)* is the scheme to assign *r*N* number of policies with highest weight as the candidate for *LPPE*. 2) Random Assignment *(RA)* is the baseline scheme, where each policy is randomly selected with equal chance to be the candidate for *LPPE*. In this scheme, *r*N* policies are randomly chosen and deployed in the line cards. The *WA* scheme performs better that the *RA* scheme and it is proved in the following Theorem.

Theorem 1: There are *N* policies $\{R_1, R_2, \ldots, R_N\}$ in the policy set, λ is the request rate and each policy rule R_i has weight assignment W_i, where $W_1 > W_2 > \ldots > W_n$ and $\sum_{i=1}^{n} W_i = 1$. With K = *r*N* policies deployed at the line card, the *WA* scheme is better than the *RA* scheme in terms of successful policy matching rate at *LPPE*.

Proof: *1)* Calculating the number of successful policy matches in the line card.

For the *WA* scheme, the *K* policies with the largest weights are deployed in the line card and the number of successful matches is given by $\lambda(\sum_{i=1}^{k} W_i)$. For the *RA* scheme, the number is given by $\lambda K / N$.

2) Derive that $\lambda(\sum_{i=1}^{K} W_i) \geq \lambda K / N$

Assume that $\lambda(\sum_{i=1}^{K} W_i) < \lambda K / N$. Then there must exist a $W_i < 1/N, i \in [1, K]$. On the other hand, $\lambda(\sum_{i=1}^{K} W_i) > \lambda(N-K)/N$. This means there exists a $W_i > 1/N, i \in [K+1, N]$. As W_i is sorted, there is a contradiction. Q.E.D.

2) Performance/Price Analysis

To make the analysis feasible, we make following assumptions: 1) The incoming packets follow *Poisson* arrival rate λ, which contributes to the packet queuing delay in both *LPPE* and *SPPE*. 2) The packet transmission delay between *LPPE* and *SPPE* is constant. The system has *K* line cards and packet arrivals at each line card are also *Poisson* with a mean rate of λ / k. 3) Packet pre-processing time in the line card and supervisor card are exponentially distributed with means of $1/\mu_1$ and $1/\mu_2$. 4) The packet delay in the middleware box include the packet matching and execution time for the matched policy, the packet transmission delay between the *LPPE* and *SPPE* and the queuing delays in the *LPPE* and *SPPE*. The packet matching time is determined by the policy size and policy complexity defined by the policy dimension. The policy execution time is also constant. 5) We assume that the cost (price) of *PPE* is the main price of the whole system.

With the increasing number of policies and dimensions, more memory and *CPU* resources are required for policy matching and processing. The price function is defined as follows [19][20].

$$C = G(N, D), \tag{1}$$

where *N* is the number of policies, *D* is the dimension of the policy rule. The price is given by, $C = G(N, D) = K_0 * N^{D/2}$, where K_0 is a constant.

By using the *M/M/1* queuing model in Figure 4, the average delay can be expressed as

$$\begin{aligned} T_d &= \left(\frac{r(1-b)}{u_1 - ((r\lambda)/k)} + d_1\right) + \left(\frac{rb}{u_1 - ((r\lambda)/k)} + d_1 + \frac{rb}{u_2 - \lambda(1-r) - rb\lambda} + d_2\right) \\ &\quad + \left(\frac{1-r}{u_2 - \lambda(1-r) - rb\lambda} + d_2\right) \end{aligned} \tag{2}$$

$$d_1 = F(rN, D) \tag{3}$$

$$d_2 = F(N, D), \tag{4}$$

where T_d is the total delay, d_1 and d_2 are delays for packet policy matching in *LPPE* and *SPPE* (note that *SPPE* has the whole policy table and *LPPE* has $r*N$ policies.) respectively; $r \in [0,1]$ is the Policy distribution Degree, b is an adjustable parameter which defines the probability of finding a match in the *LPPE*, $F(x, y)$ represents the packet delay for policy matching and policy execution and is given by $K_2*y*log(x)$ (where, x and y represent the policy number and policy dimension respectively. We consider the optimal search time for the table with size x is $log(x)$ and each policy with y dimensions needs $O(y)$ search time). The three components of equation (2) are the processing delay and queue delay for packet processing in *LPPE*, *LPPE*/*SPPE*, and *SPPE*, respectively.

From (1), we can calculate the price of system as follows:

$$C_1 = KG(rN, D) \tag{5}$$

$$C_2 = G(N, D) \tag{6}$$

$$P = (C_1 + C_2)/C_2 \tag{7},$$

where C_1 is the price of line cards, C_2 is the price of supervisor card, K is the number of line cards, $G(x,y)$ is the price function defined in (1) and the P is the comparative price compared to the standard price for the centralized architecture.

Combined with (2) and (7), the *Performance/Price (PP)* metric is defined as the achieved performance with the price spent and is given by,

$$PP = \frac{1}{T_d * P} \tag{8}.$$

We note that it is natural that the performance may not increase in the same order as price increase for some complex systems. The simple version of *PP* provides general relationship between the price and performance benefit for the analysis purposes in this paper. More complex price-benefit estimations can be a part of future work.

B. Policy Partition

1) Problem Definition

To achieve the distributed policy processing, a portion of policies need to be deployed at the *LPPE*. In order to guarantee the correctness of policy execution for the policy distribution, the policies at the *LPPE* should not have any conflict or overlapping with other policies at both the *LPPE* and *SPPE*. We define the following problem **- Policy Partition-capable Set (PPCS) Generation Problem**: Given policy set C outputted by algorithm X, each rule in set C is a 2-tuple {P_i: <C_i, A_i>, $n \geq i \geq 1$}, where P_i is the entity of policy rule, C_i denotes the Condition, and A_i

denotes Actions. M_i is the K-dimensional rectangular defined by condition C_i. We say that algorithm X solves this problem if 1) Each M_i is a rectangular-related subset in T, where $T = M_1 \cup \ldots \cup M_n$. 2) M_i and M_j have no overlap, i.e., $M_i \cap M_j = 0$, whenever $i \neq j$.

When the policy set S is generated as PPCS, we can partition the policy set S into several different smaller sets denoted by $S_1,\ldots,S_m$, where $S_1 \cup \ldots \cup S_m = S$. Thus, each S_i can be deployed separately while guaranteeing the correctness of policy execution. In other words, this policy set with PPCS can easily meet the requirements for distributed policy processing mentioned above.

The importance of PPCS is demonstrated by following example: Company network X has two smaller policy subnets namely S_1 and S_2, where $S_1 \cap S_2 = S_{12}$. We assume that company has the following ACL and IPSEC policies deployed in particular middleware box by the policy agent:

1. From S_1 To X allow only port 100-200
2. From S_2 To X allow port 100-400
3. From X to S_1 deny port 100-500
4. From X to S_2 deny port 100-1000
5. From S_1 to S_2 encrypt all ports
6. From S_1 to S_2 authenticate all ports

If policies 2, 3 and 6 are deployed at line card locally in the policy distributed processing, there might be the following issues: i) S_{12} accessing to X with port 300 will be allowed, which has the conflict with policy 1. ii) The X accessing to S_{12} with port 600 will be allowed, which has the conflict with the policy 4. iii) Packets from S_1 to S_2 are only to be authenticated without encryption. It actually violates the security of requirement policies, as the policy 5 needs the packets from S_1 to S_2 to be encrypted. So, we can say that if the incoming packet matches with several policies, then all of them must be treated together as the 'best matching policies' according to defined policy requirements. If one policy is in the policy set of LPPE, and the one is in policy set of SPPE, the inconsistency may happen. There is no way for the LPPE to know that packet has to go to the SPPE if the matching has been found locally. With the increasing number of policies for different services, this issue becomes worse.

In the next section, we will design the detailed algorithms to achieve the PPCS.

2) Policy Partition Algorithms

The basic idea of policy partition algorithm is to check the relationships among policies and generate the policies which do not have overlaps and conflicts. In order to partition the policy set, we look at the relationship between policy rules: P_i: <C_i, A_i > and P_j: <C_j, A_j >. We need consider both the condition and action tuples for the policies. The detailed description is listed:

Condition Relationship: We define several operations to describe the relationships between hyper-rectangular regions M_i and M_j constructed by conditions C_i and C_j:

i) R1 - Complete disjointed relationship: $\forall i \neq j, M_i \notin M_j \& M_j \notin M_i$;

ii) R2 - Strict superset or subset relationship: $\forall i \neq j, M_i \in M_j \mid M_j \in M_i$;

iii) R3 - Equal relationship: $\forall i \neq j, M_i = M_j$;

iv) R4 - Partial overlapping relationship: $\forall i \neq j, M_i \cap M_j \neq 0$.

Action Relationship: We assume that both action A_i and A_j are single action. We define following possible relationships between two actions A_i and A_j.

i) R5 - No Order rule: $A_i o A_j = A_j o A_i$;

ii) R6 - Order rule: $A_i o A_j \neq A_i o A_j$;

iii) R7 - Cancellation rule: $A_i o A_j = NULL$;

iv) R8 - Inclusion rule: $A_i o A_j = A_i \mid A_i o A_j = A_j$;

o represents the operation between the two actions. We state that actions operation can be applied if they satisfy above rules.

In the distributed policy processing environment, policy partition task can happen at the system initial time and system run-time. Our algorithms need to handle both situations.

Static Partition Set Generation: The procedure for the policy partition with no conflicts needs to be conducted during the system initial time. The task is to generate a *PPCS* policy set by given a non-*PPCS* policy set. The detailed algorithm is described as in Figure 5. The basic idea of this algorithm is to conduct the condition verification for relationships with R_1-R_3, we can apply the four rules – R_5-R_8 to construct the *PPCS*. For the actions with overlapping relationship – R_4, a new rule is added to cover the overlapping region and construct the *PPCS*. In this algorithm, two basic functions *CheckRelationship* and *ResolveConflict* are used (detailed description is provided in Appendix A) to verify the condition relationship to eliminate conflict resolution. Note that our algorithms can always transform the policy set into PPCS policy set. However, the number of output policies in PPCS may be larger than the one of input policies.

The time complexity for functions *CheckRelationship* and *ResolveConflict* are both *O(D)*, where *D* is the policy dimension. The worst case time complexity of the function

ConstructPartitionCapablePolicySet is *O(N(N+C)D/2)*, where *N* is the size of policy table and *C* is number of policies which are overlapping.

In Algorithm 1 shown in Figure 5, following two partition schemes are considered: 1) *BPPS (Basic Policy Partition Scheme):* It checks only the *condition* relationship between policies, i.e., equal, subset/superset or overlapping. Thus, new policies are generated and all the generated policies are disjointed. 2) *OPPS (Optimized Policy Partition Scheme):* It is easy to see that the output policy set by *BPPS* has some redundant policies, the policy *action* rules R_5-R_8 are used to compress the policy set and make the number of output policies decrease as much as possible.

```
Algorithm 1: ConstructPartitionCapablePolicySet (B, C, Enhance)
    // Input: B - input policy set with #B policies, Enhance= true means that BPSS algorithm is
    //     configured
    // Output: C - output policy set with #C policies
    // Function: go through all the entries in the policy table to check the policy relationship and
    // resolve conflict. If the new policy is generated and the system is configured with BPSS
    // optimization mode, the new added policies need perform the same procedure again with
    //  existing policies
        1.    initiate a new set F = empty
        2.    C + = B[1]
        3.    for i = 2 to #B
        4.        F = B[i]
        5.        for j = 1 to #C
        6.            G = B[j]
        7.              flag = CheckRelationship (F, G)
        8.              temp = ResolveConflict(F, flag)
        9.              if Enhance is true &&  a new policy is generated at ResolveConflict
        10.                 Based on $R_5$- $R_8$, verify any compression can be done with
                              adding new policy to exist policies
        11.             C += temp
        12.   End
```

Figure 5: Algorithm for the Policy Partition to remove conflicts

To better understand algorithm 1 with *BPPS* and *OPPS* schemes, we illustrate the following example: Assume $R_1 : <C_x , A_1>$, $R_2 : <C_y , A_2>$, $R_3 : <C_z , A_3>$ are three policies, where $C_x= C_1 | C_4 | C_6 | C_7$, $C_y= C_2 | C_4 | C_5 | C_6$, and $C_z= C_3 | C_5 | C_6 | C_7$ denote the conditions shown in Figure 6.

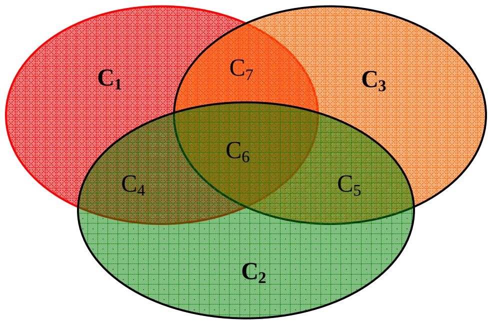

Figure 6: Example of Policy Partition

We also assume that the policy rules - A_1, A_2, and A_3 satisfy following operations, i,e., A_1 *o* $A_2 = A_1$, A_1 *o* $A_3 = A_1$ and A_2 *o* $A_3 = A_2$. By using above algorithm, a new policy rule will be added for the overlapping condition. After calculating all the overlapping regions, the policy partition-capable set with 7 policies is listed as follows by the *BPPS* scheme:

$$C_1 => A_1, C_2 => A_2, C_3 => A_3, C_4 => A_1\&A_2,$$
$$C_5 => A_2 \& A_3, C_6 => A_1 \& A_2 \& A_3, \text{ and } C_7 => A_1 \& A_3 \quad (9)$$

By using the action operation rules among A_1, A_2, and A_3, we get

$$C_1 => A_1, C_2 => A_2, C_3 => A_3,$$
$$C_4 => A_1, C_5 => A_2, C_6 => A_1 \text{ and } C_7 => A_1 \quad (10)$$

By using the *OOPS* scheme, we achieve following final result:

$$C_1 \mid C_4 \mid C_6 \mid C_7 => A_1, C_2 \mid C_5 => A_2, C_3 => A_3 \quad (11)$$

Some examples of deleting operations are listed as follows:

1) When the policy R_1 is deleted, the output rule set is

$$C_2 \mid C_5 => A_2, C_3 => A_3 \quad (12)$$

2) When the policy R_2 is deleted, the output rule set is

$$C_1 \mid C_4 \mid C_6 \mid C_7 => A_1$$
$$C_3 => A_3 \quad (13)$$

3) When the policy R_3 is deleted, the output rule set is

$$C_1 \mid C_4 \mid C_6 \mid C_7 => A_1$$
$$C_3 \mid C_5 => A_3 \quad (14)$$

Dynamic Partition Set Generation: During the system run-time, the application has some policies to be applied (add/ remove) to the middleware box and this procedure needs to run again to guarantee the new policy set to be *PPCS*. We want the policy update have minimal impacts on run-time packet processing. One practical way is for middleware box to be implemented as a dual mode with two policy processing engines: one module is active and the other module is at standby mode. When the policy update is done, the standby module conducts the policy update and *PPCS* generation without impact on the active mode. When the policy partition task is done, the standby module takes over to be the active and active module becomes the standby module. The basic idea is similar to the Algorithm 1. The detailed description is in Appendix A. The worst case execution times of *AddNewPolicy* and *DeletePolicy* algorithms are $O(D(N+C))$ and $O(D(N+C))$, respectively.

V. Simulation Results and Analysis

In this section, we present results from the evaluation of our system. We first discuss our experimental model before presenting our results.

A. Experimental Model

- *Performance Metrics:* In this paper, we propose a distributed architecture for policy processing and algorithms for policy partitioning and distribution. We evaluate the performances of the centralized and the distributed architectures using the Delay defined in Formula (2) and Performance/ Price metric (PP), defined in Formula (8). We also evaluate the performance of the policy distribution schemes using the PP metric. We use the number of increased policies and total execution time metrics to study the performance of policy partitioning. As this paper is focusing on the single middleware box, we do not consider the bandwidth cost for the policy distribution and propagation delay for the policy distribution.
- *Evaluation Systems:* Our simulation system includes a middleware box including *K* line cards and *1* supervisor card. Each subsystem is simulated by a M/M/1 system based on input rate and capacity. The *SPPE* in the supervisor card has the full copy of the policies table and each *LPPE* in the line card locally has *PDD* times the number of policies. The evaluation setups are listed below: 1) The tuple $<K, PDD, W, N, D, \lambda, \mu_1, \mu_2>$ defines the configuration parameters: *K* denotes the number of system line cards, *PDD* denotes the Policy Distribution Degree, $W \in \{WA, RA\}$ denotes whether *WA* (Weighted Assignment) or *RA* (Random Assignment) scheme is adopted for policy distribution, *N* denotes the total number of policies, *D* denotes the policy dimension, λ, μ_1, and μ_2 denote the policy request rate (*Poisson* Process), capability of line card and capability of supervisor card respectively. We will use '*' for a parameter in the tuple representation when it is not be used in the simulation scenarios. 2) To evaluate the *PP* metrics for *WA*

and *RA* schemes, we randomly generate 10000 three dimensional policies and each dimension is generated based on normal distribution (*N(300,300), mean=300 and variance=300, respectively*). 3) To evaluate the performance of policy partition algorithms, $n \in [1000, 100000]$ number of two-dimensional policies (each dimension is generated by normal distribution (*N(300,300)* and 40 policy *actions* are uniformly generated in [1,40]).

- *Evaluation Method:* We use simulation to obtain performance data.

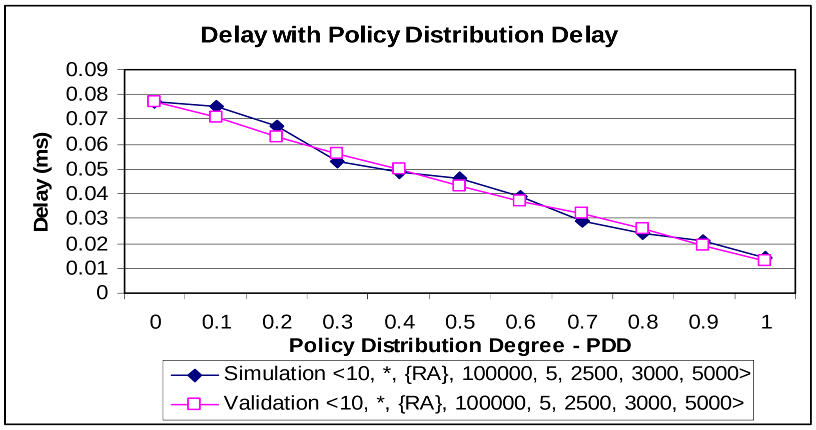

Figure 7: Delay with varying Policy Distribution Degrees

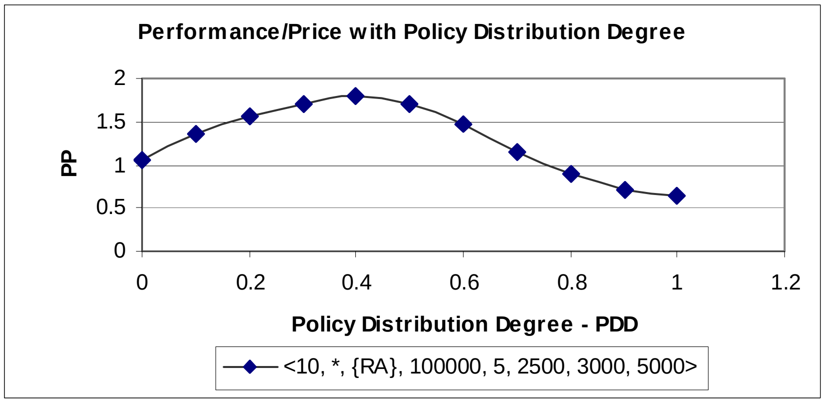

Figure 8: PP with varying Policy Distribution Degrees

B. Performance Results and Observations

Due to the space limitations, we present only a limited number of cases here. However, the conclusions we draw here generally hold for many other cases we have evaluated.

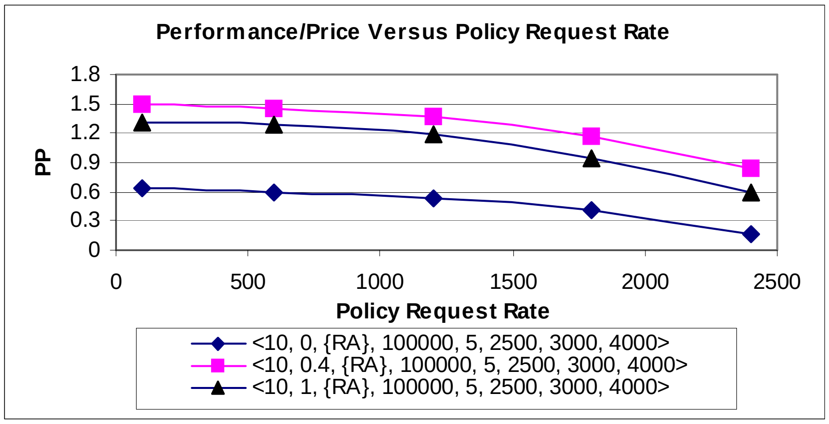

Figure 9: PP with different PDD under varying Policy Request Rate

1) Performance for Different Policy Distribution Degrees

We present the delay and PP results while comparing different architectures by varying the *PDD* in Figure 7 and 8. The system configuration here is given by *<10, *, {RA}, 5000, 5, 2500, 3000, 4000>*. In the legend of Figure 7, validation shows the validation results provided by Formula (2). We make the following observations.

- When *PDD* is set to 0, the system is purely centralized. When the *PDD* is set to *1*, the system is purely distributed. As expected, the delay in centralized system is longest and the delay is decreasing with the increase of the PDD. Both the cases result in poor performance of PP. Although the centralized architecture has smaller price, it suffers from the bottleneck of a single centralized *SPPE* resulting in low *PP*. The totally distributed architecture results in lower processing delay with higher price, resulting in a poor *PP*. Thus both the extremes are not desirable. In the following, we report our performance results. The data obtained from analytical derivations in Section IV.A are in very good agreement with our simulation data.
- *PP* increases first and then decreases with increasing *PDD*. This is due to the trade-offs as discussed above due to which both extremes result in poorer performance. Thus a carefully chosen *PDD* will result in high values of *PP*. In this particular configuration, when *PDD* is 0.4, *PP* is 1.8, which is 70 percent and 300 percent higher compared to the purely centralized and pure distributed architectures respectively.

2) Performance Comparison versus Policy Request Rate

We present results while comparing different architectures (different *PDD*) under varying policy request rate, λ in Figure 9. The system configuration is given in the Legend in Figure 9. We make the following observations:

- PP decreases with increase in λ. Also the fall in *PP* is steeper as λ increases. This is due to the fact that as the request rate becomes larger it causes longer queuing delays, resulting in poorer performance.
- Both the extreme cases of purely centralized and purely distributed architectures perform poorly. A system in between the two (*PDD = 0.4*) performs better than both extreme cases.

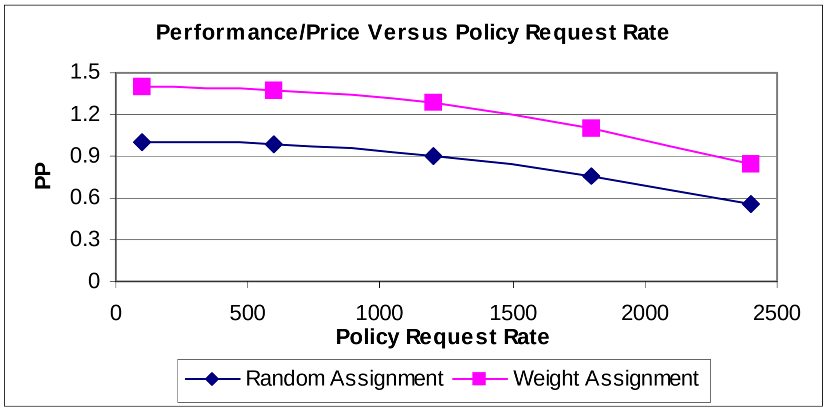

Figure 10: PP with different policy distribution schemes under varying Policy Request Rates

3) Performance Comparison of Different Policy Distribution Schemes

Figure 10 shows the performance for different policy distribution (*WA* and *RA*) schemes under varying policy request rates. The system configuration is *<10, 0.5, {RA* or *WA}, 10000, 3, *, 3000, 4000>*. Here, we make following observations.

- The performance of *WA* scheme outperforms that of *RA* scheme. As shown in *Theorem 1*, more packets can find matches in the line card locally by using the *WA* scheme, which makes the overall packet processing delays smaller and achieves higher *PP*. *PP* decreases with the increase in λ for both schemes. As expected, with increase in policy request rate queuing delays are longer, resulting in poorer *PP*.

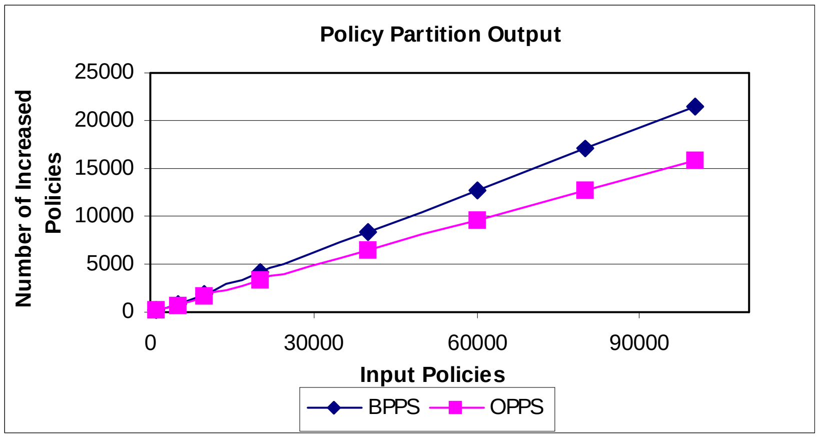

Figure 11: Increase in Output Policy Number for different Policy Partition Algorithms

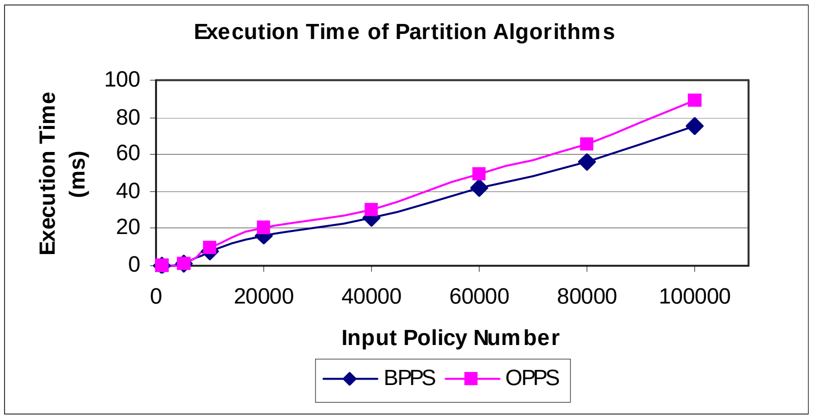

Figure12: Execution Time of different Policy Partition Algorithms

4) Performance of Policy Partition Algorithms

Due to our policy partitioning algorithms, the total number of output policies increases. Figure 11 shows this trend. Figure 12 shows the total execution times for the two partitioning algorithms (BPPS and OPPS) we propose in Section IV.

- As expected, with the number of input policies increasing, there are more chances of overlapping policy *conditions* and action conflicts. Thus, newer policies have to be generated. We argue that by using the policy compressing technologies, the number of increased policies can be effectively reduced.
- With increasing number of input policies, the execution time of algorithms also increases. As the *OPPS* needs to check the policy actions and optimize the number of output policies, it requires more time for constructing the policies.

VI. Conclusions and Future Work

In this paper, we propose a generic active-service based infrastructure. We integrate the policies requested by applications and middleware solutions to transparently execute services in heterogeneous networks. The two key components of the infrastructure are the policy agent and middleware box. We propose scalable policy processing schemes in the middleware box. To the best of our knowledge, this is the first study addressing this issue from this aspect. In summary, our technology includes the following: 1) Generic active-service based infrastructure. 2) Distributed policy processing in the middleware box. We conduct extensive performance evaluations on different architecture and algorithms. The evaluation results show that the distributed architecture can achieve over *70* percent increase of performance/price ratio with proper policy distribution degree compared to a purely centralized approach. The experimental results also demonstrate that to make the policy be both partition-capable and conflict-free, newer policies have to be generated, which can be effectively reduced by using policy compression schemes.

Our work has broad impacts. With a tremendous spurt in newer and customized services demanded by Internet applications like VPN, NAT, access control, content delivery services, defense services against attacks etc., it is increasingly difficult to deploy such services in today's existing Internet. Such services can be easily deployed with our approach presented in this paper. Another benefit of our approach is the ease in which service policies can easily change during system run time without any modifications to the existing infrastructure. To summarize, our infrastructure is general enough and a wide spectrum of customized services can be deployed with it.

There are several directions to extend our study. As the middleware box is the generic functional entity to provide different services, our design approach can easily be adopted in other areas like, high performance firewall systems, content delivery gateways etc, which will be part of our future work too.

References

[1] G. Agarwal, R. Shah, and J. Walrand, "Content *Distribution Architecture using Network Layer Anycast*", in Proceeding of IEEE Military Commnication Conference (Milcom), Washington DC., Oct. 2001.

[2] D. L. Tennenhouse and D. J. Wetherall, "*Torwards an Active Network Architecture*", In Proceedings of DANCE (DARPA Active Networks Conference & Exposition), San Francisco, CA, May 2002.

[3] C. C. Zou, W. B. Gong, and D. Tonsley, "*Code Red Worm Propagation Modeling and Analysis*", In Proceedings of the 9-th ACM conference on Computer and communications security, Washington DC. November 2002

[4] K. Calvert and S. Bhattacharjee, "*Directions in Active Networks*", IEEE Communications, *vol.36*, No. 10, Oct. 1998, pp. 72–78.

[5] M. Hicks and J. T. Moore, "*Experiences with Capsule-based Active Networking*", In Proceedings of DANCE (DARPA Active Networks Conference & Exposition), San Francisco, CA, May 2002.

[6] M. Fiuczynski and B. Bershad, "*An Extensible Protocol Architecture for Application-Specific Networking*", in Proceedings of the 1996 Winter USENIX Conference, San Diego, CA., January 1996.

[7] D. Scott, W. Arbaugh, M.Hicks, and J. Moore, "*The SwitchWare Active Network Architecture*", IEEE Network Special Issue on Active and Controllable Networks, vol. 12 no. 3, 1998.

[8] M. Hicks, A. Keromytis, and J. Smith, "*A Secure Plan*e", In Proceedings of DANCE (DARPA Active Networks Conference & Exposition), San Francisco, CA, May 2002.

[9] S. Bhattacharjee and K. Calvert, etc, "*CANEs: An Execution Environment for Composable Services*", In Proceedings of DANCE (DARPA Active Networks Conference & Exposition), San Francisco, CA, May 2002.

[10] E., Amir, S. McCane, H. Zhang, "*An Application Level Video Gateway*", In Proceedings of the third ACM international conference on Multimedia, San Francisco, CA, May 1995.

[11] S. Subramanian, P. Wang, etc, "Practical *Active Network Services within Content-Aware Gateways***",** In Proceedings of DANCE (DARPA Active Networks Conference & Exposition), San Francisco, CA, May 2002.

[12]David R. Cheriton and Mark Gritter, **"***TRIAD: A Scalable Deployable NAT-based Internet Architecture*", Technical Report, Standford University, 2001.

[13] K. L. Calvert, J. Griffioen, B. Mullins, L. Poutievski, and A. Sehgal, "*Secure, Customizable Many-to-One Communication*", In Proceedings of International Working Conference on Active Networking, Lawrence, Kansas, October 2004.

[14] S. Subramaniam, E. Komp, M. Kannan, and G. J. Minden, "*Building a Reliable Multicast Service based on Composite Protocol for Active Networks*", In Proceedings of International Working Conference on Active Networking, Lawrence, Kansas, October 2004.

[15] M. Waldvogel, G. Varghese, and J. Turner, "*Scalable High-Speed Prefix Matching*", ACM Transactions on Computer Systems, Volume 19, Number 4, November 2001.

[16] P. Warkhede, S. Suri, and G. Varghese, "*Fast Packet Classification for Two-Dimensional Conflict-free Filters*", In Proceedings of Annual Joint Conference of the IEEE Computer and Communications Societies(Infocom), Anchorage, Alaska, March 2001.

[17] P. Gupta and N. McKeown, "*Packet Classification on Multiple Fields*", In Proceedings of Annual Conference of the Special Interest Group on Data Communication (ACM SIGCOMM), Cambridge, MA, September 1999.

[18] A. McRae, "*High Speed Packet Classification*", In Proceedings of AUUG National Conference, September 1999.

[19] C. Chao, J. Ping, and K. Xu, "*A Fast IP Classification Algorithm Applying to Multiple Fields*", In Proceedings of IEEE International Conference on Communications, Helsinki, Finland, June 2001.

[20] D. Shah and P. Gupta, "*Fast Updates on Ternary-CAMs for Packet Lookups and Classification*", In Proceeding Hot Interconnects VIII, San Jose, CA, October 2000.

[21] H-W. Braun, "*Models of Policy Based Routing*", RFC 1104.

[22] Cisco Systems Inc., "*Policy-based Routing*", http://www.cisco.com/warp/public/cc/techno/protocol/tech/plicy_wp.htm

[23] Motorola Inc., "*Edge Routing Solution*", http://broadband.motorola.com/nis/edge_routing.html

[24] H. Adiseshu, S. Subhash, and P. Guru, "*Detecting and Resolving Packet Filter Conflicts*", In Proceedings of Annual Joint Conference of the IEEE Computer and Communications Societies(Infocom), Tel-Aviv, Israel**,** March 2000.

[25] M. Debasis and J. A. Morrison, "*Virtual Private Networks: Joint Resource Allocation and Routing Design*", In Proceedings of Annual Joint Conference of the IEEE Computer and Communications Societies(Infocom), New York, NY, March 1999.

[26]T. Leong, P. Shor, and C. Stein, "*Implementation of Combinatorial Multicommondity Flow Algorithms*", In Proceeding of International Conference on Integer Programming and Combinatorial Optimization, Houston, Texas, June 1998.

[27] K. Geihs, "*Middleware Challenges Ahead*", IEEE Computer, Vol 34, No. 6, pp24-31, June 2001.

[28] Z. S. Chen, L.X. Gao, and K. Kwiat, "*Modeling the Spread of Active Worms*", In Proceedings of Annual Joint Conference of the IEEE Computer and Communications Societies(Infocom), San Francisco, CA, March 2003.

[29] N. Provos, "*A Virtual Honeypot Framework*", In Proceedings of 12-th Usenix Security Symposium, Boston, MA, August 2004.

[30] M. Bailey, E. Cooke, F. Jahanian, J. Nazario, and D. Watson, "*The Internet Motion Sensor: A Distributed Backhole Monitoring System*", In Proceedings of 12-th Annual Network and Distributed Systems Security Symposium, San Diego, CA, Febrary 2005.

[31] J. Aweya, "*IP Router Architecture: An Overview*", Nortel Network Technical Report, 1999.

[32] H. C. Chan, H. M. Alnuweiri, and V. C. Leung, "*A Framework for Optimizing the Cost and Performance of Next-Generation IP Router*", IEEE Journal on Selected Areas in Communication, Vol. 17, No. 6, June 1999.

[33] L. Kleinrock, "*Queueing System*", Vol-1., New York, Wiley, 1975.

Appendix A:

Basic Functions for Policy Partition:

```
Algorithm 2: AddNewPolicy(F, B, C)
        // Input: F - new policy and B - policy set with #B policies
// Output: C – PPCS policy set
// Function: Check the relationship between F and the entity of policy set B; based on the
// CheckRelationship result and use the ResolveConflict function to resolve the policy
// confliction.
1.  initiate a new set C = null
2.  for i = 1 to #B do
3.      flag = CheckRelationship(F, B[i])
4.      ResolveConflict (F, B[i], flag)
5.      C += temp;
6.  return C
7.  End
```

```
Algorithm 3: DeletePolicy( F, B, C)
  // Input: F - new policy and B - Input policy set with #B policies
  // Output: C – PPCS policy set
  // Function: check the relationship between F and all the entity in set B; based on the
  // CheckRelationShip results, different actions are conducted, such as, if overlapping, a new
  // rule need to retained.
  1.  for i = 1 to #B
  2.      flag = CheckRelationship (F, B[i])
  3.      if flag is 'equal'
  4.          delete B[i] from B
  5.      if flag is 'subset/superset'
  6.          apply the rules $R_5 – R_8$ to generate new rule X and overwrite rule B[i]
              by X
  7.      if flag is 'overlapping'
  8.          generate the two rule $X_1$ and $X_2$, which corresponds to the overlapping
              region and non-overlapping region;
              delete the rule B[i] and insert new rule {$Y_1$ , $Y_2$ } to C
  9.  return C
  10. End
```

```
Function 1: CheckRelationship ($Y_i$, $Y_j$)
  // Input: policies $Y_i$ and $Y_j$
  // Outupt: flag={disjoint, superset/subset, equal, overlapping}
// Function: check the relationship between two rules $Y_i$ and $Y_j$, where $Y_i$ and $Y_j$ are tuples <$C_i$,
// $A_i$> and <$C_j$, $A_j$>.  $M_i$ and $M_j$ are the hyper-rectangles constructed by $C_i$ and $C_j$, respectively.
    1.    if $M_i$ and $M_j$ are disjoint
    2.        return 'disjoint'
    3.    if $M_i$ and $M_j$ are subset and superset
    4.        return 'superset/subset'
    5.    if $M_i$ and $M_j$  are equal
    6.        return 'equal'
    7.    if $M_i$ and $M_j$  have intersection
    8.        return 'overlapping'
    9.    End
```

```
Function 2: ResolveConflict ($Y_i$, $Y_j$, flag)
    // Input:  policies $Y_i$, $Y_j$ and  flag={disjoint, superset/subset, equal, overlapping}
    // Function: Solve the rule with conflict between $Y_i$, $Y_{j.}$ based on the flag value returned by
    //  CheckRelationship function call. According to the flag setting, the conditions of original
    //  policies are compared. If there is an overlapping between conditions of the policies, the new
    // policy will be generated.
    1.    if flag is 'disjoint' return set {$Y_i$, $Y_j$}
    2.    if flag is 'superset/subset'
    3.        check the $A_i$ and $A_j$, apply rules {$R_5$-$R_8$}, and generate new action rules
    4.        with { $R_{n1}$, $R_{n2}$} and return
    5.    if flag is 'equal'
    6.        check the Ai and Aj, apply four rules {$R_5$-$R_8$}, generate new rule -
    7.        < $A_n$, $R_n$> and return
    8.    if flag is 'overlapping'
    9.        For m = 1 to k do
    10.       let $X_i$ be the longer of two prefix $C_i$[m] and $C_j$[m]
    11.           generate new policy set R = <C, A>,
    12.           where C = <$X_1$, …., $X_k$> and A = $A_i$ U $A_j$
    13.            apply rule {$R_5$-$R_8$}, construct F' and G' to take out C from the
                   overlapping region
    14.       return new policy set with {F', G', R}
    15.   End
```
