Establishing PDCA Cycles for Agile Network Management in SDN/NFV Infrastructure

Kotaro Shibata†, Hiroki Nakayama‡, Tsunemasa Hayashi‡, Shingo Ata†
† Graduate School of Engineering, Osaka City University
3–3–138 Sugimoto, Sumiyoshi-ku, Osaka 558-8585, Japan
Email:{shibata@c., ata@}info.eng.osaka-cu.ac.jp
‡ BOSCO Technologies Inc.
1–4–12 Nishishinbashi, Minato-ku, Tokyo 105–0003, Japan
Email:{nakayama, hayashi}@bosco-tech.com

***Abstract*—Software Defined Networking (SDN) is attracting many researchers in networking area, especially in the field of network operations and managements. A fine-grained network management that controls traffic dynamically by the unit of flows is one of the key challenges towards a resilient network for the future Internet. To this end, the network is required to re-configure timely, adaptively and dynamically. In this paper, we propose a self-adaptive approach to manage both network and functional resources in SDN. To achieve the quick adaptation to various pattern of traffic, we first establish a Plan-Do-Check-Act (PDCA) cycle for our network management architecture. Then we describe a detailed procedure including interactions from networking components. We also show some use cases and give a proof-of-concept deployment.**

***Keywords*—SDN, NFV, Agile Network Management, Network Control, Optimization, Security**

I. INTRODUCTION

Software Defined Networking (SDN) is attracting many researchers in networking area, especially in the field of network operations and managements. SDN enables separating control planes from data planes. It is able to configure dynamically managed resources, not only construction of virtual networks but also development of networking functions (NFVs).

A fine-grained network management that controls traffic by the unit of *flows* is one of the key challenges towards a resilient network for the future Internet. An application flow is composed of some session flows. When we manage these session flows instead of application flows, it is expected to improve QoE and to keep flows secure, to name but a few. E.g., in a case of security attacks such as malicious traffic, it is required that a very quick detection and its effective measure to keep the managed network securely and safety. However, such network operations are usually expensive because they need complicated processing of forwarding packets on data plane. Under resource-constrained network, there is optimization problem to achieve the highest efficient configurations from the perspective on the use of a given managed resource. Since this efficiency strongly depends on the policy to control, managed network requires to re-configure timely, adaptively and dynamically in terms of short period [1], [2], [3], [4].

To catch up dynamic traffic changes rapidly and adaptively, there are numbers of related works presented so far. [5] presents Stratos, the design and an implementation of the orchestration layer for virtual middle boxes. Stratos enables to make efficient and correct composition of virtual middle boxes in the presence of dynamic scaling via SDN mechanisms. [6] proposes Procera, which is a network control framework to express event-driven network policies using a high-level functional programming language. [7] proposes a new framework called CellSDN to support many subscribers, frequent mobility, fine-grained measurement and control, and real-time adaptation in cellular networks. [8] proposes a new system called Dionysus, which achieves fast and consistent network updates in SDN. [9] proposes FlowGuard to conduct automatic and real-time violation resolutions with the help of several innovative resolution strategies designed for diverse network update situations. [10] proposes FlowTags, which introduces Tags for systematic policy enforcement.

Despite above related works, we still need more general framework to support dynamic adaptation not only in terms of traffic but also of policy because a new policy definition is based on what happened in the network a-priori. Furthermore, re-optimization of traffic control for a given policy must be triggered by events from network monitoring. Mediation of multiple different policies is also a challenging problem in adaptive policy management. In this paper, we propose a self-adaptive approach to manage both network and functional resources in SDN. To achieve the quick adaptation various pattern of traffic to complicated policies, we consider a whole policy as a combination of small and quick adjustable sub-policies. We then establish a Plan-Do-Check-Act (PDCA) cycle for our network management architecture. After that, we describe a detailed procedure including interactions from networking components. We also show some use cases and give a proof-of-concept deployment.

We show the outline of our proposed architecture in Section II and the details in Section III. We then describe the implementation of our architecture in Section IV. Section V provides our proof-of-concept model, experimental evaluations and discussions. Finally, we conclude our paper with future research topics in Section VI.

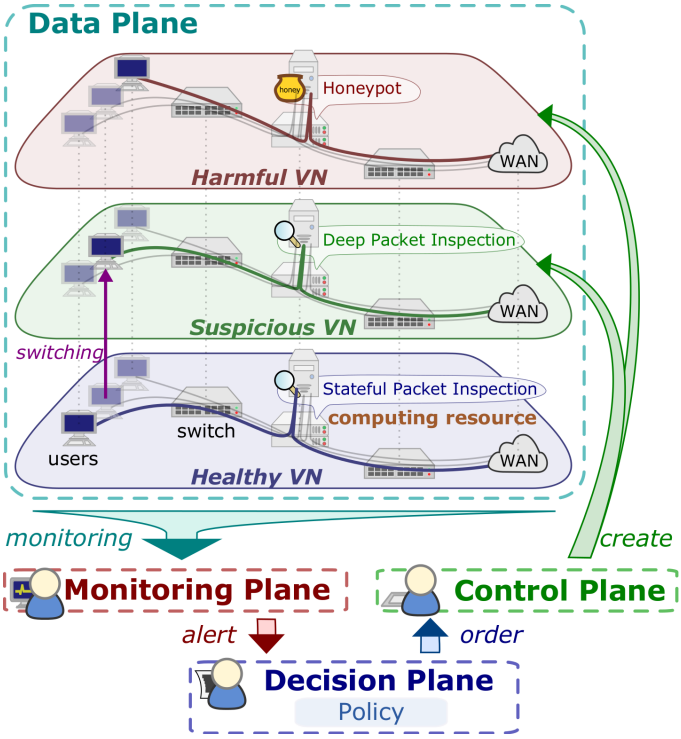

Fig. 1. Overall Architecture of Proposed Infrastructure

II. Proposed Architecture

A. The Concept

Figure 1 shows the overview of our proposed architecture. In this figure, SDN switches are deployed to construct a managed network that supports network virtualization. Furthermore, there is a computing resource pool that connects to the managed network to provide on-demand networking functions. Computing resources is a hosting server where networking functions are running over virtual machines individually. End terminals are connected to one of the SDN switches to join the managed network. Some SDN switches are connected to Wide Area Network (WAN) for communications to global network.

In our design, there are three management planes. Monitoring Plane is used to collect information about managed resources. After statistical analysis, Monitoring Plane may raise an alert to Decision Plane. Decision Plane has to solve an optimal set of configurations to perform an objective function according to the policy, under the limited amount of physical resources and the constrained countermeasures. We isolate a decision process from Control Plane. This is because the decision process is a key to self-reconfiguration and is more complicated rather than general routing algorithms. Furthermore, solving algorithms can be varied in the policy. Decision Plane is preferable to be isolated from other planes, to be replaced by another one easily. Control Plane is used to control managed resources actually by sending control messages to controllers based on the output from Decision Plane.

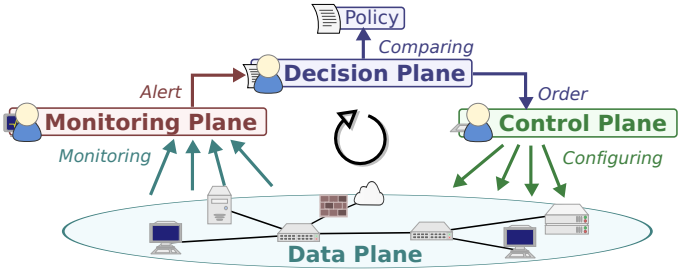

Fig. 2. PDCA Cycle in Management Part

B. Establishment of PDCA Cycles for Self-Adaptation

Our architecture requires a quick reconfiguration of managed resources to adopt traffic change promptly. For this end, we conduct a PDCA Cycle (See Figure 2). PDCA Cycle is usually referenced for agile development of new idea with incremental deployment including feedback. According to the PDCA Cycle, our management procedures can be divided into following four phases.

1) **Problem Identification (Plan Phase):** This phase is initiated by an alert message from Monitoring Plane. The problem related the alert is clearly stated by Decision Plane referring to the policy.
2) **Solving Problem and Control (Do Phase):** After the problems are analyzed, a set of configurations is sent to Control Plane. Then the controller configures managed resources actually.
3) **Checking Configurations and Feedback (Check Phase):** After the configurations are executed, it is verified what the decision is properly executed by feedback information from Monitoring Plane. After that, the controller configures a new alert for the next phase.
4) **Alert (Act Phase):** This phase has two types of processing. Reactive processing is aimed to keep a good achievement in given policy. In proactive processing, a new policy is defined to improve various types of metrics. An administrator usually makes the definition of a new policy.

C. Policy Slice

For dynamic and adaptive policy management, policy should be able to implement easily and incrementally. We consider a whole policy as a composition of mini-policies, *policy slice*. *Policy slice* is a minimum portion of the whole policy that can run a PDCA cycle independently. A *policy slice* is responsible to manage a specific part of traffic according to a policy specialized for a target traffic.

Multiple *policy slices* drive parallel PDCA cycles simultaneously, as shown in Figure3. *Policy slice* has alert parameters, decision algorithms and countermeasure configurations. Alert parameters are used to configure alerts in Monitoring Plane. Decision algorithms are performed in Decision Plane resolving a problem related the alert. Countermeasure configurations are used to generate a template of network and virtual function assignments. A suite of validation for the decision can be included optionally.

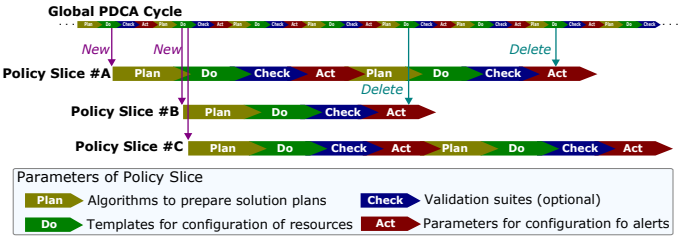

Fig. 3. Policy Slices and Global PDCA Cycle

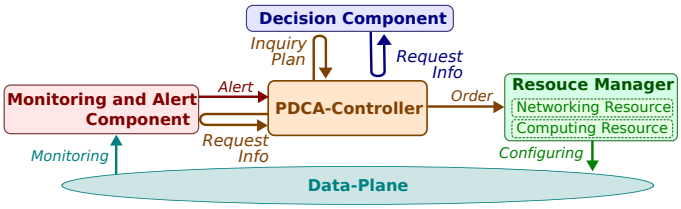

Fig. 4. System Components in Proposed Architecture

If multiple *policy slices* are running independently, some *policy slices* may interfere each other, which causes a negative impact to the whole infrastructure. To address this problem, we establish a global PDCA cycle for the management of *policy slices*. The global PDCA cycle focuses on administrative and strategic issues that include addition/update/deletion of *policy slices*. When an incident has occurred, the administrator needs to configure a new policy to take effective measures to it. In the global PDCA cycle, Plan phase decides to create policy slices for collected incidents that have risen before the Plan phase. In the Check phase, all *policy slices* are reviewed in term of interference and overall efficiency. After that, some *policy slices* may be merged and deleted. The others are kept with slightly adjusting parameters. All actions to *policy slices* are performed in the Act phase of the global PDCA cycle.

III. Details of Architecture Components

In this section we describe four major system components to realize the PDCA cycle, as shown in Figure 4.

A. PDCA-Controller

PDCA-Controller is a component to drive both cycles of the global PDCA and *policy slices*. PDCA-Controller also provides an abstraction of functions of the other components to achieve commodity and interoperability. PDCA-Controller has four states that are associated to PDCA phases.

1) **Plan State:** After an alert, PDCA-Controller prepares a solution *plan* against the alert. The alert has come from Monitoring and Alert Component with parameters, time, type, severity and priority. The alert is forwarded to Decision Component via PDCA-Controller. When multiple related alerts are simultaneously raised, PDCA-Controller may filter or aggregate alerts to avoid duplicated actions in Decision Component. PDCA-Controller can be requested to report the current status of managed resources. These requests are forwarded to Monitoring and Alert Component and the replies are sent back to Decision Component. After receiving the *plan* from Decision Component, PDCA-Controller verifies the reply and transits to Do state.
2) **Do State:** According to the *plan*, PDCA-Controller sends a set of requests to control and optimize managed resources actually. The state is moved to Check state after all configurations have been ready.
3) **Check State:** PDCA-Controller verifies what the *plan* is actually taken and whether the *plan* actually solves the problem or not. Then PDCA-Controller moves to Act state.
4) **Act State:** According to Act Phase, PDCA-Controller performs both reactive and proactive processing. In reactive processing, PDCA-Controller configures a new alert for the next PDCA cycle of individual *policy slice*. The proactive processing is performed for the global PDCA cycle that includes re-organization and re-composition of *policy slices*. Furthermore, additional strategic or experimental *policy slices* may be installed manually for further improvement.

B. Monitoring and Alert Component

Monitoring and Alert Component is responsible for the management of all monitoring information by a central control.

1) **Monitoring devices:** Monitoring information includes statistics and performance metrics collected from all managed resources via monitoring agents. This component also sends probe messages for verifying availability. Collected information is managed by the monitoring database to retrieve easily the appropriate information against a request.
2) **Alert:** An alert is generated when managed resources satisfy given conditions. To this end, this component accepts an *alert set request* with some specific parameters. Identifier is a unique value of the alerts. Target Device indicates the device to monitor. Condition is represented as a kind of expression like "load $\geq$ 0.5".
3) **Data retrieval:** Monitoring and Alert Component can provide monitoring status according to the request by specifying parameters, e.g., device and metric. Specifying a statement-like expression, e.g., average and maximum, can also retrieve some statistics.

C. Decision Component

Decision Component provides a *plan* to solve the issue associated with an alert. First an alert message is notified from PDCA-Controller that is delegate from Monitoring and Alert Component. Alert message contains the alert identifier and its severity. Decision Component selects the most appropriate solving algorithm by these alert parameters. Additional information about managed resources can be retrieved as a result of a request to PDCA-Controller. The solving algorithm is pre-configured when the corresponding *policy slice* is added. Solving algorithm is typically (1) a specialized countermeasure

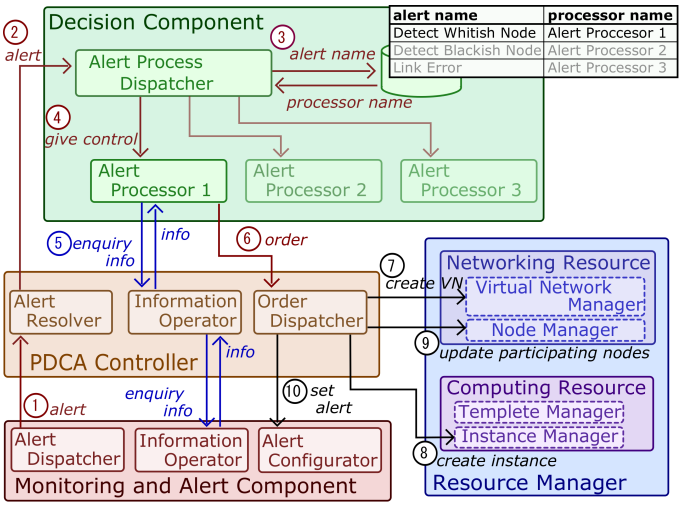

Fig. 5. Detailed Block Diagram of Functional Components

for the target alert or (2) a solution of optimization problem for a given condition. The solving algorithm finally results a *plan*, which is used to control managed resources for a specific traffic type defined by packet classification rules.

D. Resource Manager

Resource manager is a component to manage all managed resources for a traffic control. Networking resources has a capacity of virtualization to apply differentiated control to specific types of traffic. Computing resources are also assigned to create NFVs dynamically when additional processing is required. One additional feature of Resource Manager is that it can assign resources tentatively to support Check phase in PDCA cycle. In the Check phase, the target traffic is mirrored to the temporary assigned virtual resources in addition to the current assigned virtual resources.

IV. Implementation

In this section, we describe implementation issues to realize functional components. Figure 5 shows the block diagram of functional components. All APIs are RESTFul. In other words, inter-components communications are invoked by REST (Representational State Transfer) [11]. All parameters in REST APIs are given by JSON (JavaScript Object Notation) [12] format.

A. Processes and APIs

1) PDCA-Controller: PDCA-Controller has three processes, Alert Resolver, Information Operator and Order Dispatcher. Following APIs are provided.

1) **`GET /pdca-v0.1.0/info/`** Request to retrieve the current status of managed resources. Information Operator handles the API and calls an `info` API to Monitoring and Alert Component.
2) **`PUT /pdca-v0.1.0/order/`** Request a control of managed resources to execute a *plan*. The API is mainly invoked in Do State and Order Dispatcher calls configuration APIs to Resource Manager. Then it transits to Check State for the validation of *plan*.
3) **`PUT /pdca-v0.1.0/alert/`** Receive an alert from Monitoring and Alert Component. This API is mostly called in Act State. After the call, Alert Resolver handles it and calls an `alert` GET API to Decision Component. Alternatively, the administrator can manually call the API to explicitly request a decision.
4) **`GET/POST/DELETE /pdca-v0.1.0/policy-slice/`** Maintain *policy slices*. GET method is used for listing *policy slices* and for a detailed information of a specific *policy slice*. POST method creates a new *policy slice* with parameters. DELETE method deletes a *policy slice*.

2) Monitoring and Alert Component: Monitoring and Alert Component has three processes, Alert Dispatcher, Information Operator and Alert Configurator. Following APIs are provided.

1) **`GET /mac-v0.1.0/info/`** Request to retrieve the current status of managed resources. Information Operator handles the API which queries the monitoring database and returns a response.
2) **`GET/POST/DELETE /mac-v0.1.0/agent/`** Maintain monitoring agents. POST method is called to create a new agent. Parameters can be a binary of execution command or an URI of agent repository. Alert Configurator handles the API.
3) **`GET/POST/DELETE /mac-v0.1.0/alert/`** Maintain alerts. POST method is used for creating a new alert. GET method is called to get the alert list. Alert Configurator handles the API. Alert Dispatcher continuously monitors the conditions of alerts and calls `alert` API of PDCA-Controller when a condition is satisfied.

3) Decision Component: Decision Component consists of mainly two types of processes, Alert Process Dispatcher and multiple Alert Processors. Following APIs are provided.

1) **`PUT /decision-v0.1.0/alert/`** Receive an alert as a request to prepare a *plan*. Alert Process Dispatcher firstly handles it. The dispatcher has a database that associates an alert with its solving algorithm. When the alert name is found, the dispatcher executes the associated Alert Processer. The processor can call `info` API of PDCA-Controller to get information of devices. It also runs solving algorithm to return the *plan* as the response.
2) **`GET/POST/DELETE /decision-v0.1.0/solver/`** Maintain solving algorithms for Alert Processors. When a new *policy slice* is created, a new solving algorithm is installed via POST method. The parameter may be a script of solving algorithm or an URI of solving algorithm repository.

4) Resource Manager: Resource Manager has two sub components, Networking Resource and Computing Resource.

Networking Resource has multiple virtual networks.

1) **`GET/POST/DELETE /rm-v0.1.0/network/`** Maintain virtual networks. POST method is called to create a

new virtual network. GET method is used to list or get detailed information of virtual networks.

2) `GET/POST/DELETE /rm-v0.1.0/network/(netid) /node/` Maintain virtual nodes in the specified virtual network (netid). Participating nodes can be added, deleted and updated via the API. List of participating nodes can also be obtained by GET method. Virtual nodes include NFVs provided by computing resources.

Computing Resource maintains a database of *function template* and its *instance*.

1) `GET/POST/DELETE /vfp-v0.1.0/function/` Maintain *Function Templates*. Same as monitoring agents and solving algorithms. *Function templates* is created by putting a binary of function or specifying URI of template repository as a parameter.
2) `GET/POST/DELETE /vfp-v0.1.0/function /(funcid)/instance/` To create a new *instance*, call POST method with funcid which is the identifier of *function template*. Instance-specific configurations are specified by parameters.

B. Deployment of Proposed Infrastructure: An Example

We develop and deploy our proposed infrastructure in experimental environment. The whole network is constructed by connecting devices with OpenFlow switches. For Monitoring and Alert Component, we configure Zabbix, an open source distributed monitoring framework [13]. Information Operator process calls Zabbix APIs to configure and retrieve status of resource. We use OmniSphere [14] as an OpenFlow controller. Therefore, we implement processes of Networking Resources by using Omnisphere APIs. For NFV Pool, we deploy Docker [15] to use lightweight virtual machine appliances. We implement all APIs and related internal processes of four components by Python.

V. Experimental Evaluation

In this section, we verify how our proposed architecture works and evaluate the effectiveness of our approach.

A. Experimental Scenarios

Initially, all traffic flows are sent out to the Internet via a firewall directly (See Scenario #1 in Figure 6). After a couple of seconds, an administrator finds some traffic including anomalous traffic that seriously degrades reliability of the managed network. The administrator then decides to forward all traffic flows to an Intrusion Detection System (IDS) to detect and monitor anomalous traffic (See Scenario #2). The PDCA-Controller creates a new *policy slice* (1) to deploy IDS in NFV pool, and (2) to connect the IDS node between the OpenFlow switch and the firewall. As a result, all traffic flows are detouring to go along IDS. However, since the IDS requires a huge computational overhead, the output rate from the IDS is significantly lower than the one from the firewall. Therefore, the rate of outbound traffic to the Internet degrades significantly which leads a remarkably under-utilization of the link to the Internet. Such performance degradation seriously affects the quality of experience to end-users.

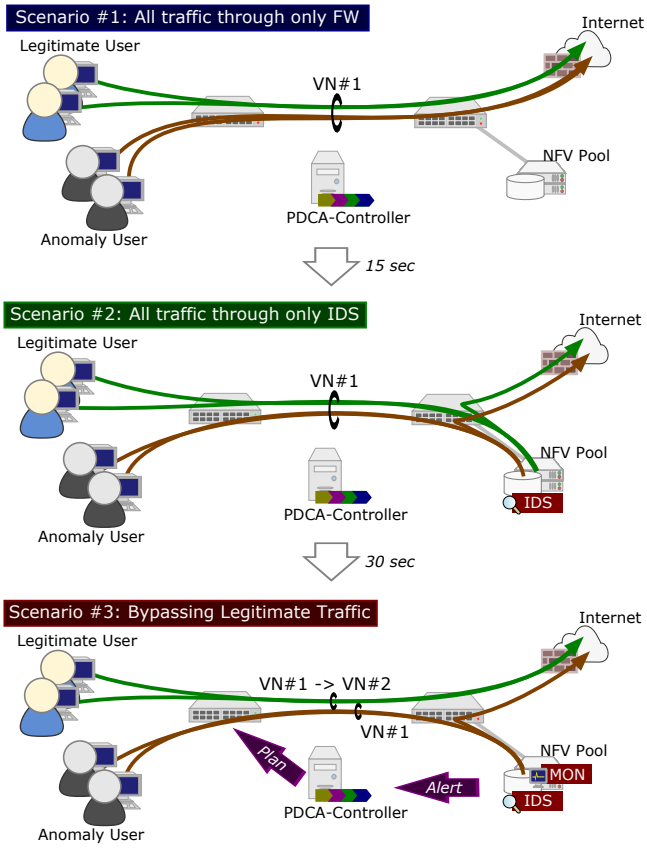

Fig. 6. Experimental Environment

After the careful analysis of the traffic, the administrator finds reliable legitimate traffic mixed in the aggregated traffic. If the administrator can separate such legitimate traffic from the whole, these legitimate traffic flows can be passed directly to the firewall without processing by the IDS. Finally, it would improve the performance of both total output rate and the transmission delay of legitimate traffic. From this analysis, the administrator decides to implement a strategy to treat the legitimate traffic bypassing from the IDS. For this purpose, a new *policy slice* is created (1) to install a rule to detect the legitimate traffic, (2) to create a *plan* to bypass the IDS and (3) to configure the alert for legitimate traffic detection (See Scenario #3).

B. Experimental Setup

To evaluate the above scenario, we conducted experimental evaluation by following steps.

1) We setup 10 virtual clients who upload legitimate contents and 10 virtual clients who upload anomalous one. Both contents are uploaded to the file server beyond the firewall.
2) Each virtual client uploads only one file simultaneously and repeats to upload immediately after the previous

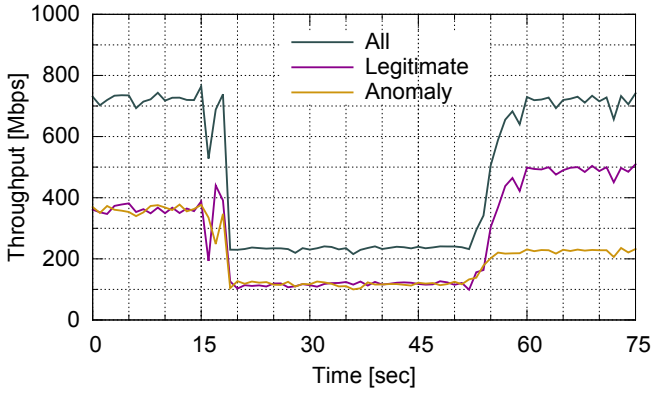

Fig. 7. Variation of Traffic in All Scenarios

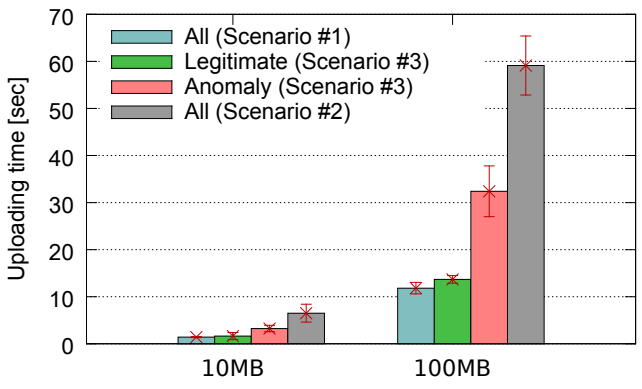

Fig. 8. Comparisons of Uploading Time

uploading.

3) The size of uploaded file is varied from 10 MB to 100 MB.
4) The IDS is deployed as an appliance of virtual machine. We use Suricata [16] as an IDS example.
5) The IDS identifies traffic flow whose first 23 bytes of main content in HTTP are equal to "This is a good content" as legitimate one.
6) PDCA-Controller switches the virtual network for legitimate traffic flows from VN #1 to VN #2 after the alert from the IDS.
7) We create new *policy slices* for Scenarios #2 and #3 after 15 and 45 seconds respectively.

C. Results and Discussions

Figure 7 shows the variation of traffic in all scenarios. The graph has three plots, throughput of legitimate users, anomalous users and total users. All throughputs are measured at the link between the OpenFlow switch (right side) and the firewall. The result between 0 sec to 15 sec is for Scenario #1, 15 sec to 45 sec for Scenario #2 and 45 sec to 75 sec for Scenario #3. First, the total throughput is over 700 Mbps at the initial stage. Both legitimate and anomalous traffic has similar throughputs (around 350 Mbps). At approx. 4 sec later from the installation of the *policy slice* for Scenario #2, the total throughput is suddenly degraded to around 250 Mbps. This is because the IDS output performance is around 250 Mbps. In other words, the IDS becomes the bottleneck. Second, after the installation of a *policy slice* for Scenario #3 at 45 sec, both throughputs of legitimate and anomalous traffic are gradually increasing. Unlike sudden decrease in the case of Scenario #2, the increase is gradual and it takes approx. 15 secs to converge. This is not due to delays for the transition of PDCA cycle. The new *policy slice* is applied to newly generated flows after 45 sec. Note here the average overhead of the virtual network creation is approx. 162 msec with 4.25 msec in standard deviation. The average delay of flow table updates is around 148 msec with 3.36 msec in standard deviation. We consider such overheads are sufficiently small to achieve a quick response to apply *policy slices*. Finally, in Scenario #3, the throughput of anomalous traffic can reach close to 250 Mbps. It indicates a new *policy slice* for Scenario #3 effectively works. This is because the IDS doesn't process legitimate traffic. Furthermore, the total throughput is also close to 750 Mbps almost same performance as in the case without IDS. It indicates that we achieve high throughputs and reliability, while all anomalous traffic is filtered by the IDS.

Figure 8 compares the average uploading time among all scenarios. In Scenarios #1 and #2, both legitimate and anomalous contents have similar delays to upload. However, They are differentiated in Scenario #3. Error bars are also added to show 5 and 95 percentiles of delays. In this figure, the uploading time is significantly increased due to the processing of the IDS, which reaches about 6x longer than the case when we only use the firewall. On the other hand, in Scenario #3, legitimate traffic can benefit a good performance. The overhead is less than 10%. The effect of bypassing legitimate traffic has also a good influence to anomalous traffic because the IDS can be involved more for processing anomalies traffic. As a result, even the IDS processing anomalous traffic, the delay can be reduced around 50% compared to Scenario #2.

VI. Conclusion

In this paper, we have proposed a novel self-adaptive approach to manage both network and functional resources in SDN. We have proposed a new framework to establish a Plan-Do-Check-Act (PDCA) cycle for both network and computing resource management adaptively, quickly and timely. We have given both design and implementation of our infrastructure in detail to show the feasibility as well. Furthermore, we have conducted some practical scenarios as a real use case and have presented experimental results to show the high potential of our proposal. For future research topic, we need to consider other *policy slices* to solve various real use cases. Additional experiments in the large network environment are important for the validation of the scalability.

Acknowledgment

This work was supported by JSPS KAKENHI Grant Number 15H02694.

REFERENCES

[1] R. P. Esteves and L. Z. Granville, "On the management of virtual networks," *IEEE Communications Magazine*, vol. 51, no. 7, pp. 80–88, July 2013.

[2] M. F. Bari, S. R. Chowdhury, R. Ahmed, and R. boutaba, "Policycop: An autonomic QoS policy enforcement framework for Software Defined Networks," in *Proceedings of Software Defined Networks for Future Networks and Services (SDN4FNS 2013)*, Trento, Italy, November 2013, pp. 1–7.

[3] Z. Movahedi, M. Ayari, R. Langar, and G. Pujolle, "A survey of autonomic network architectures and evaluation criteria," *IEEE Communications Surveys & Tutorials*, vol. 13, no. 2, pp. 464–490, May 2012.

[4] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proceedings of the 10th USENIX Conference on Networked Systems and Design and Implementation (NSDI)*, Lombard, IL, April 2013, pp. 99–112.

[5] A. Gember, A. Krishnamurthy, S. S. John, R. Grandl, X. Gao, A. Anand, T. Benson, V. Sekar, and A. Akella, "Stratos: A network-aware orchetration layer for virtual middleboxes in clouds," March 2014.

[6] H. Kim and N. Feamster, "Improving network management with software defined networking," *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, February 2013.

[7] L. E. Li, Z. M. Mao, and J. Rexford, "CellSDN: Software-defined cellular networks," in *Proceedings of 2012 European Workshop on Software Defined Networking*, Darmstadt, Germany, October 2012, pp. 7–12.

[8] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer, "Dynamic scheduling of network updates," in *Proceedings of ACM SIGCOMM 2014*, Chicago IL, August 2014, pp. 539–550.

[9] H. Hu, W. Han, G.-J. Ahn, and Z. Zhao, "FLOWGUARD: Building Robust Firewalls for Software-Defined Networks," in *Proceedings of the 3nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Network (HotSDN 2014)*, Chicago, IL, USA, August 2014, pp. 97–102.

[10] S. K. Fayazbakhsh, V. Sekar, M. Yu, and J. C. Mogul, "FlowTags: Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions," in *Proceedings of the 2nd ACM SIGCOMM Workshop on Hot Topics in Software Defined Network (HotSDN 2013)*, Hong Kong, China, August 2013, pp. 19–24.

[11] T. Fieldings, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[12] D. Crockford, "The application/json media type for JavaScript object notation (JSON)," December 2008.

[13] "Zabbix." [Online]. Available: http://www.zabbix.com

[14] "Omnisphere." [Online]. Available: http://www.stratosphere.co.jp/product/solution/sol01.html

[15] "Docker." [Online]. Available: http://www.docker.com

[16] "Suricata." [Online]. Available: http://suricata-ids.org