NEOD: Network Embedded On-line Disaster Management Framework for Software Defined Networking

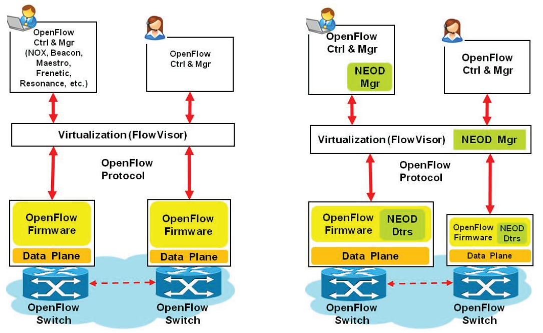
Sejun Song, Sungmin Hong, Xinjie Guan, Baek-Young Choi, and Changho Choi

***Abstract*—Network management against security, reliability, and performance attacks is an integral part of building dependable, high-performance network services. In traditional networks, the management practices take mainly *remote* approaches to cope with the ossified network infrastructure. However, since disaster events that occur within the network should be inferred by the remote management system on the network edge, the problems are often accumulated and enlarged, and the diagnosis is delayed, inaccurate, unreliable, and not scalable. Although a few embedded approaches become available, they are costly and limited to the vendor specific applications. Software-Defined Networking (SDN) architecture has been recently proposed in order to enable flexible network control plane. However, it has focused mainly on traffic engineering, network virtualization, and off-line configuration management problems, and there is very little research on on-line management against network disaster. In this paper, we propose a globally deployable Network Embedded On-line Disaster (NEOD) management framework for SDNs. NEOD addresses important network management issues including agility, accuracy, reliability, and scalability. We have identified critical network disaster management metrics to adaptively support versatile application requests on SDNs. We have implemented the proposed system and metrics in OpenFlow with OpenWrt based routers, and have shown the effectiveness of NEOD through extensive system experiments as well as Mininet-based simulations.**

I. Introduction

You Can't Fix What You Can't Measure: Unmanaged network failures, congestions, mis-configurations, and security attacks are the common causes of on-line network disasters and directly relate to network security, reliability, and performance. For example, a network failure can be caused either by actual faults or deliberate security attacks. If it is not detected and managed timely, it will degrade network availability and may eventually cause service discontinuation. The purpose of network disaster management is primarily to determine the root cause in *real-time* in order to isolate disaster within the contained area. It, in turn, monitors the customer Service Level Agreement (SLA) [1] conformance, assesses and identifies weak areas for network improvement, and predicts potential security attacks. Hence, network disaster management is an integral part of building dependable, high-performance network services.

Dr. Sejun Song and Sungmin Hong are at Texas A&M University, College Station, Xinjie Guan and Dr. Baek-Young Choi are with University of Missouri-Kansas City, and Dr. Changho Choi is at Cisco Systems.


OpenFlow Ctrl & Mgr (NOX, Beacon, Maestro, Frenetic, Resonance, etc.)
OpenFlow Ctrl & Mgr
Virtualization (FlowVisor)
OpenFlow Protocol
OpenFlow Firmware
Data Plane
OpenFlow Switch
NEOD Mgr
NEOD Dtrs

(a) OpenFlow (b) NEOD in OpenFlow

Fig. 1. OpenFlow and NEOD Architectures

You Could Have Done It So Much Better: Despite the effort by the research community as well as network operations to build effective on-line network disaster management tools, there is a lack of real field deployed systems that subscribe to the purpose. In the traditional networks, the network disaster management practices take mainly *remote* approaches to cope with the ossified network infrastructure. However, the *remote* approach is ineffective for the following reasons:

- **It does not respond rapidly:** Since disaster events that occur within the network should be inferred by the remote management system on the network edge via polling, notification, and logging, the diagnosis is delayed and the problems are often accumulated and enlarged.
- **It is not scalable:** Since the remote event polling consumes both system and network resources, it is not scalable when the network size or the number of monitored components increases.
- **It is not reliable:** Since the management message can be lost in the presence of link failures and router crashes, it is not reliable.
- **It is not accurate:** Due to the system and performance limitations, the remote management cannot detect detailed disaster events within the network devices.

A few embedded approaches such as Cisco's Embedded Event Manager (EEM) [2] and Cisco's Component Outage On-Line (COOL) Measurement [3], [4] become available. However, they are costly and limited to the vendor specific devices. Software-Defined Networking (SDN) architecture has been recently proposed in order to enable flexible network control plane, [5], [6]. SDN decomposes networks using the network distribution, forwarding, and configuration abstractions in order to support open development environments, efficient network virtualization, and management and control cost reduction. Particularly, fueled by increasing data center networking and cloud computing industries, SDN has been building up significant momentum toward the production network deployment. Being a detailed embodiment of the SDN architecture, as illustrated in Figure 1(a), OpenFlow [5], [7], [8] extracts the control and management planes out from the network devices to the remote controller and enables the current router architecture to allow flexible traffic forwarding rules. A lot of research with SDN has mainly focused on issues of traffic engineering with various remote controllers such as NOX [9], [10], Beacon [11], Trema [12], and Maestro [13], [14], the network virtualization with FlowVisor [15], [16], and the off-line configuration programming and management with Resonance [17], [18], [19], SNAC (Simple Network Access Control) [20], OMNI [21], Frenetic [22], and NetCore [23]. They fundamentally take the remote approach, and are vulnerable to various disaster events. Meanwhile, SDN opens up a unique opportunity for the network embedded management practices. However, there is very little research on the on-line management against network disasters.

Let's Bell the Cat: In this paper, we propose a globally deployable network embedded on-line disaster management framework named *NEOD*. We investigate various network service aspects including agility, accuracy, reliability, and scalability in order to provide a practical network disaster management system. As presented in Figure 1(b), NEOD is designed on an OpenFlow Switch as an OpenFlow firmware extension that does not require vendor specific changes to deploy it. NEOD supports 1) *abstraction layers* that provide a vendor-agnostic detector deployment and facilitate a dynamic policy configuration; 2) *a two-tier* system approach in which NEOD detectors, in a switch, perform light-weight disaster event detection & filtering, and NEOD manager, in a controller, performs network-wide disaster correlation and detector management. It reduces performance impact on the network device, provides scalability, and ensures dynamic deployment.

Furthermore, we identify network disaster events that are particularly ineffective and vulnerable in OpenFlow networks including *new flow attacks*, *interface flapping*, and *event storm*. We present the effective solutions with embedded event detectors to ensure that those disasters are detected, prevented, and filtered in real-time. For instance, since OpenFlow moves the control plane to a remote controller, a new flow decision should be done by the remote controller. We show that this remote control makes the OpenFlow networks vulnerable to new flow packet attacks. We propose a switch embedded light-weight event detector that analyzes the trend of the CPU utilization. It provides initial indication and information for preventing disaster events in real-time.

The rest of the paper is organized as follows. Section II

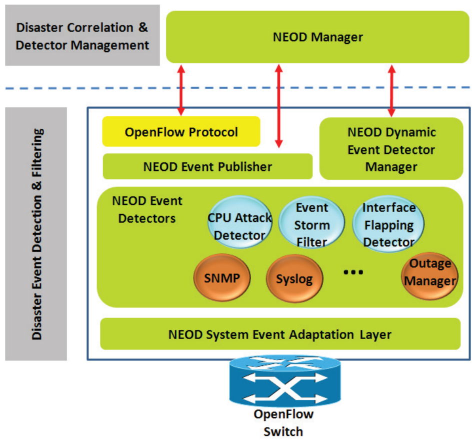

Disaster Correlation & Detector Management
NEOD Manager
Disaster Event Detection & Filtering
OpenFlow Protocol
NEOD Dynamic Event Detector Manager
NEOD Event Publisher
NEOD Event Detectors
CPU Attack Detector
Event Storm Filter
Interface Flapping Detector
SNMP
Syslog
...
Outage Manager
NEOD System Event Adaptation Layer
OpenFlow Switch

Fig. 2. NEOD Architecture

presents the proposed NEOD management framework architecture and the disaster event detectors in detail. The system implementation and evaluation of NEOD are presented in Section III. We conclude the paper in Section IV.

II. Network Embedded On-line Disaster (NEOD) Management Framework

In this section, we first explain the proposed NEOD management framework architecture in detail, and then present three disaster event detectors.

A. NEOD Management Framework Architecture

From the architectural point of view, NEOD is based on a two-tier framework. As shown in Figure 2, it consists of two functional segments, a disaster event detection & filtering segment and a disaster correlation & detector management segment. In a disaster event detection & filtering segment, NEOD is embedded in an OpenFlow Switch as a light-weight OpenFlow firmware extension. It consists of NEOD Event Detectors, NEOD Dynamic Event Detector Manager, NEOD Event Publisher, and NEOD System Event Adaptation Layer. NEOD Event Detectors include basic event handlers such as SNMP, Syslog, and Outage manager as well as application specific handlers such as a CPU Attack Detector, an Event Storm Filter, and an Interface Flapping Detector. NEOD Event Detectors monitor the disaster events according to the configured policies. The raw data are persistently maintained within the switch, and are sent to or polled by the NEOD Event Publisher. The NEOD Event Publisher selects OpenFlow protocols, SNMP MIBs, or Syslogs to communicate with the remote NEOD manager. The NEOD Dynamic Event Detector Manager dynamically downloads disaster management policies from the NEOD Manager in the controller and adaptively executes them in the OpenFlow switch as NEOD Event Detectors. This dynamic deployment capability reduces performance impact on the network devices and provides scalability to the NEOD Event Detector management. The NEOD System Event Adaptation Layer facilitates a vendor independent environment to simplify the event detector deployment as well as dynamic configuration to filter vendor critical information.

In a disaster correlation & detector management segment, NEOD Manager can reside in one or more controllers. Using the network topology, routing, and configuration information from the controllers as well as the disaster event information from the individual switches, NEOD Manager can perform network-wide disaster event correlation for the applications such as verification of customer's SLA and DoS attack detection. The NEOD management framework further enables an accurate root cause classification and a detailed event prediction that have been considered as not scalable or impossible to conduct. For example, facilitating a CPU utilization measurement on each individual router can provide a potential indication of abnormal events such as a DoS attack. Traditional approaches are mainly based upon a watchdog to set a threshold (i.e., instantaneous CPU utilization is above 90%). However, this indication alone cannot propose dependable information to predict disaster events such as a DoS attack. Hence, in practice, the ability to handle abrupt events in real-time is a very difficult issue. Instead of choosing a remote or embedded approach, the NEOD management framework harmonizes both approaches. While a light-weight embedded system analyzes the trend of the disaster and responds rapidly to the disaster on the source of the problem, a remote system performs network-wide correlation.

B. Disaster Event Detectors

1) The Case Against New Flow Attacks: Since OpenFlow moves the control plane from an individual switch to a remote controller, unlike the traditional network devices, an OpenFlow switch needs more effort to handle unknown incoming packets. As illustrated in Figure 3, when a flow packet arrives on an OpenFlow switch, it checks the flow table. If the flow packet is not in the flow table, it should send a new flow request to the remote controller via a secure channel (SSL). The OpenFlow controller handles the new flow request using the policies and routing information. It sends a decision to the OpenFlow switch via a secure channel (SSL). According to the new flow decision, the OpenFlow switch adds the new flow information into its flow table or drops the new flow packet. In practice, adversaries can inject randomly generated ***New Flow*** packets into an OpenFlow switch port to ***Attack*** the switch (named ***New Flow Attack***). The OpenFlow switch needs to consult the remote controller for each unknown flow packet via a secure channel. It may, however, cause control and data overheads to saturate CPU usage in the switch instantaneously. One of the main objectives of OpenFlow is the network virtualization so that new features can be deployed on the operating network. A FlowVisor is designed to ensure the resource isolation within the OpenFlow network. However, we have identified several New Flow Attack scenarios that the remote FlowVisor cannot handle properly. We will further present the scenarios in Section III.

To ensure the CPU resource isolation against a New Flow Attack, we propose a switch embedded light-weight CPU usage detection function. As shown in Algorithm 1, the CPU usage detector periodically (for a slot time interval (STI)) checks CPU usages of both the OpenFlow data path module ($OFdatapath$) and the OpenFlow control path module ($OFprotocol$). It saves the number of clock ticks *Jiffy* and the observed time T for the time slot i. A sliding window WS is used for a CPU usage calculation. For every STI, an average CPU usage is calculated using an accumulated $Jiffy$ with a period time WS (Average CPU Usage = (*Jiffy* for WS) / WS). If the CPU usage of a port is over the threshold Th_p, the port drops incoming packets for the time interval PDI.

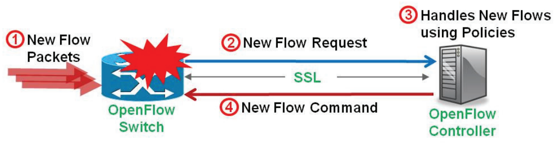


Fig. 3. New Flow Attack

Algorithm 1 NEOD CPU Isolation Algorithm
```
1:  for every $STI$ for both $OFdatapath$ and $OFprotocol$ do
2:      read current Jiffy and $Time$;
3:      advance both $cur\_index$ and $first\_index$ by 1;
4:      write $J[cur\_index]$ and $T[cur\_index]$;
5:      read $J[first\_index]$ and $T[first\_index]$;
6:      $CPUusage = (J[cur\_index] - J[first\_index]) / (T[cur\_index] - T[first\_index])$;
7:      if $CPUusage \geq Th_p$ then
8:          drop the incoming packet from the port $p$ for $PDI$;
9:      end if
10: end for
```

2) The Case Against Interface Flapping: Unlike the traditional networks, a simple switch status change may cause various cascading actions in an OpenFlow network. For example, as shown in Figure 4, when an OpenFlow switch detects a port failure, it will send a port-status message to the controller. The controller checks its policy, topology, and routing information to find the flows that use the failed port. The controller recalculates the alternative flows and sends flow modification messages to the switches in order to update the flow forwarding tables. In practice, hundreds of logical interfaces can be configured for a port and each interface may be used by many different flows. If a port fails, it will also cause failures to all the logical interfaces configured on the port. The switch will send hundreds of status change messages to a controller. A port connected to the failed port on the other switch may also send hundreds of status change messages. Being the single place of holding the network meta knowledge, the controller needs to handle all the received port status messages in a short time period. It checks its policy, topology, and routing information for each request as well as updates all the information at the same time. Considering many related flows need to be updated, it may

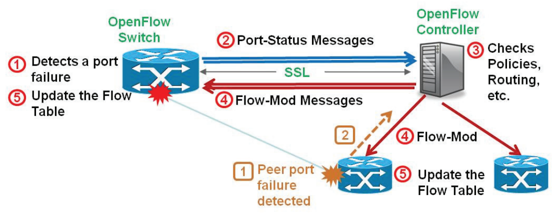


Fig. 4. OpenFlow Status Change Scenarios

also send out thousands of flow modification messages to the switches. As all the exchanged messages are encrypted, the overhead on the controller, switch, and network cannot be trivial. Furthermore, if the status keeps on changing, it may cause a significant problem on the OpenFlow network. Indeed, it is not uncommon to see that the status of an interface object keeps changing between up and down due to certain transient problems such as misconfiguration or partial physical failures. It is also possible that an adversary can keep on causing up and down situations intentionally. This is called *interface flapping*. This unstable condition should be detected rapidly and should get the operator's attention in time. Otherwise, it may cause critical disasters especially in an OpenFlow network due to the related significant overheads.

To prevent the problems caused by the frequent status changes, we propose a light-weight switch embedded interface flapping detection function. As described in Algorithm 2, when a set of down and up events is detected within FI_Th, the interface flapping detector starts to count the number of flapping events FC. If the FC exceeds the configured flapping count threshold FC_Th during the flapping interval FI, it is considered as a flapping condition. The interface flapping detector sends a $Flapping\_Start$ notification to the network management. It may take follow-up actions such as to mark the flapping interface as a logically down status until the unstable condition is resolved. Once FC becomes less than FC_Th, a $Flapping\_End$ notification is sent to indicate that the flapping condition has been resolved.

3) The Case Against Event Storm: The network objects may have a *containment relationship* among each other. For example, as illustrated in Figure 5, a router device contains many line card objects. Each line card object also contains many physical interfaces (ports). In turn, each physical interface contains many logical or virtual interfaces. If objects are in a *containment relationship*, a status change on an object causes status changes on all the objects it contains. It, in turn, may produce intensive status change notifications to cause an *event storm*. For example, a port failure may trigger thousands of logical interface failure events. With thousands of event notifications, it may cause tremendous overheads on the switch itself as well as the network and management services. As discussed in Section II-B2, if the *event storm* is not handled properly, it may cause significant problems in the OpenFlow network due to the related overheads.

To prevent the *event storm*, we propose a light-weight

Algorithm 2 NEOD Interface Flapping Detection Algorithm

```
1: if a set of down and up event is detected within $FI\_Th$ then
2:   $Fl\_Start$ is $Y$; $Fl\_Notified$ is $N$;
3:   while $Fl\_Start$ is $Y$ do
4:     reset $FC$; $FC++$ for each flapping event;
5:     wait for $FI$;
6:     if $FC > FC\_Th$ then
7:       if $Fl\_Notified \equiv N$ then
8:         send a $Flapping\_Start$ notification;
9:         $Fl\_Notified$ is $Y$;
10:      end if
11:    else
12:      if $Fl\_Notified \equiv Y$ then
13:        send a $Flapping\_End$ notification;
14:      end if
15:      $Fl\_Start$ is $N$;
16:    end if
17:  end while
18: end if
```

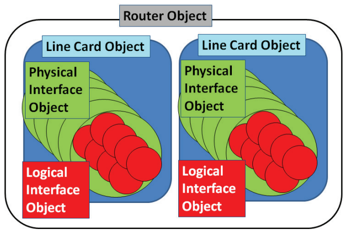


Fig. 5. Object Containment Relationship

switch embedded *event storm* filtering function. As described in Algorithm 3, when a status change event of an object is detected, the *event storm* filter checks the event drop list (EDL). If the object is already marked, the event will be ignored. Otherwise, it will check the *containment relationship*. If the object has child objects, it will be added in EDL to ignore the same status change events from the child objects. The object will be further added into the status notification list (SNL), after removing all the child objects within SNL. After an interval (ESI), port-status messages for the objects within SNL will be sent to the controller. By sending a few representative object events, we can avoid the *event storm*.

III. Evaluations

In this section, we evaluate the effectiveness and efficiency of NEOD through extensive system experiments and simulations. We implemented NEOD management framework and three critical disaster detectors in OpenFlow with OpenWrt-based Linksys WRT54GL routers [24] as well as used Mininet [25] simulation for the scalable interface flapping tests.

Algorithm 3 NEOD Event Storm Filtering Algorithm

```
1: if a down or up event of an object $X$ is detected then
2:     check $EventDropList(EDL)$ and drop if marked;
3:     check the $containment\ relationship$ of the object;
4:     if $X$ has child objects then
5:         mark in $EDL$ to ignore the same status change events
           from the child objects;
6:     end if
7:     check $StatusNotificationList(SNL)$ to remove any
       child objects of $X$;
8:     add $X$ into the notification list;
9:     send a notification for the objects within $SNL$ after
       $EventStormInterval(ESI)$;
10: end if
```

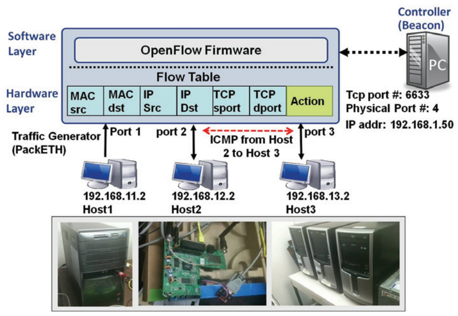


Fig. 6. New Flow Attack Test Setup

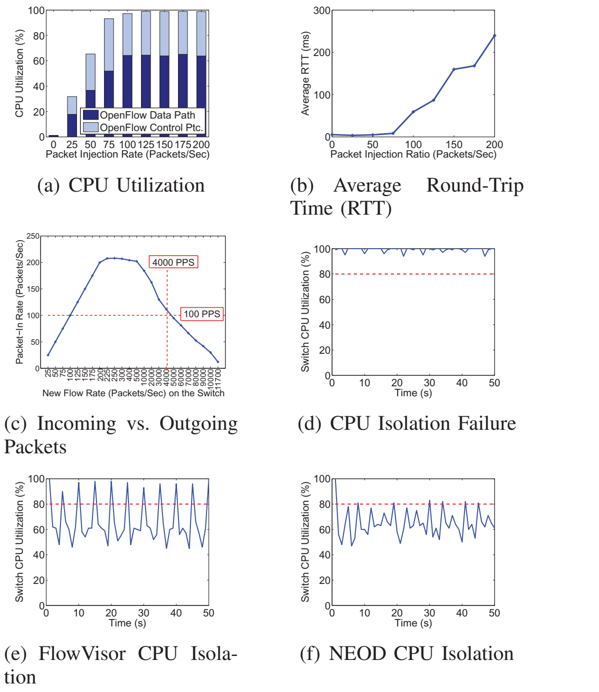


Fig. 7. NEOD Efficiently Isolates Switch's CPU While FlowVisor Cannot Fully Control Switch's CPU Utilization

A. The Case against New Flow Attacks

We have implemented a CPU resource isolation function in an OpenWrt switch to test New Flow Attack scenarios. As shown in Figure 6, three hosts and one Beacon controller are connected to the four port OpenWrt router. Host 1 uses a PackETH traffic generator [26] to inject packets into the OpenFlow switch. Hosts 2 and 3 exchange ICMP messages. We create a New Flow Attack by injecting garbage traffic into the OpenFlow switch port 1 from the traffic generator in host 1. As shown in Figure 7(a), when new flow packets are inserted into an OpenFlow switch, they cause both control and data overhead to saturate CPU usage in the switch. The CPU utilization of the OpenFlow protocol increases proportionally to the packet injection rate until the CPU is saturated. It is mainly due to the secure channel (encryption and decryption) overheads for sending new flow requests. However, it should be clearly observed that the OpenFlow protocol overhead does not exist in the traditional switches. It makes the OpenFlow switch more vurnerable to the New Flow Attack. Since there is no CPU isolation mechanism, the New Flow Attack on port 1 directly impacts the existing regular traffic between ports 2 and 3. For example, the average RTT in Figure 7(b) clearly shows that the ICMP messages between ports 2 and 3 are greatly impacted by the New Flow Attack on port 1. These results indicate that a New Flow Attack is real and easy to be created. Although OpenFlow itself does not provide the CPU isolation, FlowVisor [15], [16] remotely monitors the OpenFlow switch's new flow packet count to ensure the CPU isolation among the virtual slices. However, as presented in Figure 7(c), when the CPU is already saturated due to the abrupt injection of the new flow packets, the actual number of new flow packets sent to the FlowVisor (i.e., 100 pps) can be far less than the real incoming new flow packet counts (i.e., 4000 pps). Figure 7(a) also shows that the protocol CPU utilization stays the same after the CPU saturation (packet injection rate 100) regardless of the incoming packet rate. It indicates that only a few packets are actually sent to the FlowVisor. In this case, the FlowVisor fails to detect the critical CPU problem in the remote switch as shown in Figure 7(d). Figure 7(e) also presents that the FlowVisor cannot accurately ensure the CPU isolation for the remote switch. Although an average CPU utilization may meet the CPU isolation target (i.e., 80 %), it frequently violates the resource limitation. It is mainly due to the delayed response. These experimental results confirm that the remote resource control has intrinsic difficulties. However, as presented in Figure 7(f), the proposed embedded CPU usage detection function can control the CPU resource very accurately. It also shows that the average CPU usage is better than the FlowVisor results in Figure 7(e).

B. The Case Against Interface Flapping

We have implemented an interface flapping detection function in a Mininet simulation environment that enables the creation of a virtual OpenFlow network on a single machine. As illustrated in Figure 8, Mininet OpenFlow switches are created on a single virtual machine and Beacon is used as

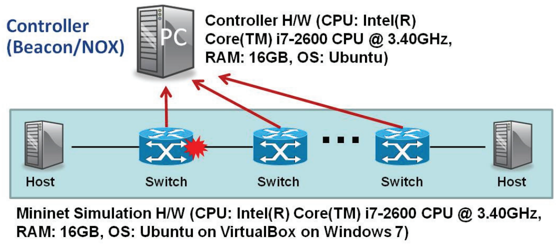
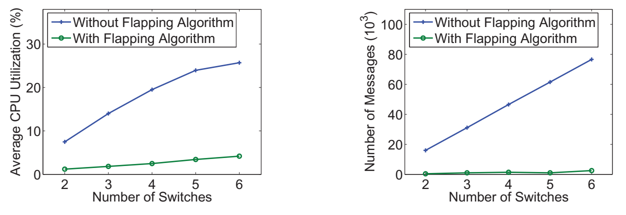

Controller (Beacon/NOX)
Controller H/W (CPU: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, RAM: 16GB, OS: Ubuntu)
Host
Switch
Switch
Switch
Host
Mininet Simulation H/W (CPU: Intel(R) Core(TM) i7-2600 CPU @ 3.40GHz, RAM: 16GB, OS: Ubuntu on VirtualBox on Windows 7)

Fig. 8. Interface Flapping Mininet Simulation Environment



(a) Controller CPU Utilization as the Number of Switches Changes

(b) Port Status Messages from Switches

Fig. 9. Comparison Average CPU Utilization and Port Status Messages Change Between Switches With and Without Flapping Detection Algorithm When the Network Scale Increases

a main controller. The Beacon controller runs on an Eclipse debug mode with applications including LearningSwitch (self learning from the new flow messages), Topology (switch liveness check with LLDP), and Routing (APSP: All Pairs Shortest Path). We implemented up to 255 logical interfaces (configurable) for a port. We created an interface flapping by changing a port status on a switch that also causes the status changes on the contained logical interfaces. First, we checked the controller CPU utilization by varying the number of logical interfaces. We used both NOX and Beacon controllers. As the controller needs to recalculate the existing flows and sends flow modification messages to the switches, we also changed the number of switches to see how the network size impacts the controller performance. As shown in Figure 10, the CPU utilization on a controller increased proportionally to the number of logical interfaces. The result shows that the controller CPU utilization became around 35% in a small network (three switches) with 255 logical interfaces for a port. It indicates that a simple port status change can cause great overhead in an OpenFlow network. We tested both the CPU utilization and the number of messages on a controller with an interface flapping detection algorithm by varying the number of switches. We used a Beacon controller with 255 logical interfaces for each switch port. As the controller recalculated the flows and sent flow modification messages, the network size impacted the controller performance. As presented in Figures 9(a) and 9(b), without the interface flapping detection algorithm, the controller CPU utilization was increased about 30% and the number of received messages was increased around 80K messages. Considering a relatively small network (six switches) was used in the experiment, it caused a significant performance overhead. However, with the interface flapping detection algorithm, the controller CPU utilization was kept at less than 5% as well as the number of messages were far less than 5K messages. The event storm filtering algorithm was applied in this experiment along with the interface flapping detection algorithm. It indicates that a simple embedded algorithm can achieve significant performance improvement especially in an OpenFlow network.

C. The Case Against Event Storm

We have implemented an event storm filtering algorithm in an OpenWrt switch. We created an event storm by changing the port status on a switch that also caused the status changes

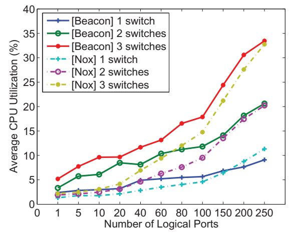


Fig. 10. NOX and Beacon CPU Utilization as the Number of Logical Interfaces Changes

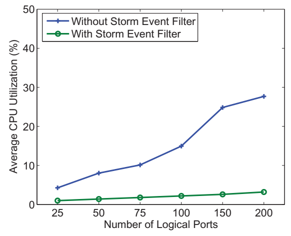


Fig. 11. Event Storm impacts CPU Utilization

on the contained logical interfaces. We checked the average CPU utilization of a switch by varying the number of logical interfaces up to 200 ports. As shown in Figure 11, the average CPU utilization becomes around 30% with 200 logical interfaces for a port without the using event storm filtering wihle it is only 3% by using the event filtering algorithm. Considering typical configurations on the type switches, it indicates that a simple port status change can cause great overhead in an OpenFlow network.

IV. Conclusion

The traditional networks, as well as the current SDN, mainly take remote approaches to network management where measured data are sent to a remote monitor or controller. We have shown that they are ineffective and vulnerable to various network disaster events using concrete examples including new flow attacks, interface flapping, and event storm. We proposed a Network Embedded On-line Disaster (NEOD) management framework that works on SDN architecture. NEOD is vendor-agnostic, and succinctly addresses the issues of agility, accuracy, reliability, and scalability with a switch embedded light-weight detector/analyzer, rendering itself as a practical network disaster management system. We have implemented the proposed system and metrics in OpenFlow with OpenWrt based routers as well as Mininet-based simulations. We have developed algorithms for detecting disaster events, and shown the effectiveness of NEOD, especially compared to the plain OpenFlow environment without NEOD.

REFERENCES

[1] D. Ameller, "Service level agreement monitor (salmon)," in *Proceedings of Seventh International Conference on Composition-Based Software Systems*, Madrid, Spain, Feb. 2008.

[2] Cisco embedded event manager (eem). [Online]. Available: http://twitter.com/ciscoeem/

[3] Component outage on-line (cool) measurement. [Online]. Available: http://www.cisco.com/en/US/docs/ios/12_0s/feature/guide/fscool12.html/

[4] J. H. Sejun Song, "Internet router outage measurement: An embedded approach," in *Proceedings of the IEEE/IFIP NOMS*, Seoul, Korea, Apr. 2004.

[5] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.

[6] J. S. Turner and D. E. Taylor, "Diversifying the internet," in *Proceedings of IEEE Global Telecommuinications Conference (GLOBECOM'05)*, Dec. 2005, pp. 755–760.

[7] *OpenFlow Switch Specification*, openflow-spec-v1.1.0.pdf, OpenFlow standard) Std., Rev. 1.1.0, Feb. 2011. [Online]. Available: http://www.openflow.org/documents/

[8] Open networking foundation (ONF). [Online]. Available: http://www.openflowswitch.org

[9] N. Gude, T. Koponen, J. Pettit, B. Pfaff, M. Casado, N. McKeown, and S. Shenker, "NOX: towards an operating system for networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 3, pp. 105–110, Jul. 2008.

[10] Nox. [Online]. Available: http://www.noxrepo.org/

[11] Beacon. [Online]. Available: http://www.beaconcontroller.net/

[12] Trema. [Online]. Available: http://http://trema.github.com/trema/

[13] Maestro. [Online]. Available: http://code.google.com/p/maestro-platform/

[14] Z. Cai, "Design and implementation of the maestro network control platform," Master's thesis, Rice University, Houston, 2009.

[15] R. Sherwood, G. Gibb, K.-K. Yap, G. Appenzeller, M. Casado, N. McKeown, and G. Parulkar, "Can the production network be the testbed?" in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, ser. OSDI'10. USENIX Association, 2010, pp. 1–6.

[16] ——, "Flowvisor: A network virtualization layer," Deutsche Telekom Inc. R&D Lab, Stanford University, Nicira Networks, Tech. Rep. OPENFLOW-TR-2009-1, Oct. 2009.

[17] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark, "Resonance: dynamic access control for enterprise networks," in *Proceedings of the 1st ACM workshop on Research on enterprise networking*. ACM, 2009, pp. 11–18.

[18] H. Kim, T. Benson, A. Akella, and N. Feamster, "The evolution of network configuration: a tale of two campuses," in *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, ser. IMC '11. ACM, 2011, pp. 499–514.

[19] N. Feamster, H. Balakrishnan, J. Rexford, A. Shaikh, and J. van der Merwe, "The case for separating routing from routers," in *Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, ser. FDNA '04. ACM, 2004, pp. 5–12.

[20] SNAC. [Online]. Available: http://snacsource.org/

[21] D. M. F. Mattos, N. C. Fernandes, V. T. da Costa, L. P. Cardoso, M. E. M. Campista, L. H. M. K. Costa, and O. C. M. B. Duarte, "Omni: Openflow management infrastructure," in *Proceedings of IEEE International Conference on Network of the Future (NOF'11)*, Paris, France, Nov. 2011, pp. 52–56.

[22] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, "Frenetic: a network programming language," *SIGPLAN Not.*, vol. 46, no. 9, pp. 279–291, Sep. 2011.

[23] C. Monsanto, N. Foster, R. Harrison, and D. Walker, "A compiler and run-time system for network programming languages," *SIGPLAN Not.*, vol. 47, no. 1, pp. 217–230, Jan. 2012.

[24] Openwrt. [Online]. Available: https://openwrt.org/

[25] Mininet. [Online]. Available: http://yuba.stanford.edu/foswiki/bin/view/OpenFlow/Mininet/

[26] packeth. [Online]. Available: http://packeth.sourceforge.net/