Management Patterns: SDN-Enabled Network Resilience Management

Paul Smith*, Alberto Schaeffer-Filho†, David Hutchison‡ and Andreas Mauthe‡
*Safety and Security Department, AIT Austrian Institute of Technology, Austria
Email: paul.smith@ait.ac.at
†Institute of Informatics, Federal University of Rio Grande do Sul, Brazil
Email: alberto@inf.ufrgs.br
‡School of Computing and Communications, Lancaster University, United Kingdom
Email: {d.hutchison, a.mauthe}@lancaster.ac.uk

***Abstract*—Software-defined networking provides abstractions and a flexible architecture for the easy configuration of network devices, based on the decoupling of the data and control planes. This separation has the potential to considerably simplify the implementation of resilience functionality (e.g., traffic classification, anomaly detection, traffic shaping) in future networks. Although software-defined networking in general, and OpenFlow as its primary realisation, provide such abstractions, support is still needed for orchestrating a collection of OpenFlow-enabled services that must cooperate to implement network-wide resilience. In this paper, we describe a resilience management framework that can be readily applied to this problem. An important part of the framework are policy-controlled *management patterns* that describe how to orchestrate individual resilience services, implemented as OpenFlow applications.**

I. Introduction

Resilience management requires the flexible configuration of network devices, such as routers and switches, in order to rapidly adapt their operation due to, for example, changing network traffic volumes or the detection of anomalous traffic profiles. In previous work [1], [2] we have demonstrated the use of policy-based resilience strategies to contain malicious attacks such as Distributed Denial of Service (DDoS), worm propagations, and other types of network challenges. These resilience strategies are based on the reconfiguration of network devices when evidence about anomalies are observed in the network. Typically large-scale resilience services, e.g., DDoS detection and remediation, are built out of the cooperation of a number of interacting devices across the network that provide more elementary services, e.g., flow monitoring, anomaly detection and traffic shaping, to name but a few.

Software-defined networking (SDN) provides a flexible architecture for quick and easy configuration of network devices. Software-defined networking in general, and OpenFlow as its primary realisation, increase routing scalability and support abstractions for traffic engineering by decoupling the switch's routing decisions (control path) from its internal packet forwarding logic (data path). This is known as the separation of the control plane from the data plane. In particular, the control plane is enforced by a remote controller, which is responsible for all routing decisions and for switch reconfiguration. However, although OpenFlow supports the abstractions needed for decoupling the data and control planes in the implementation of individual services, support is still needed for orchestrating a collection of OpenFlow-enabled devices that must cooperate to implement a large-scale service. To illustrate such a large-scale service, we describe how resilience services need to coordinate a range of detection and remediation mechanisms to combat network challenges. We thus advocate that, although SDNs and OpenFlow adequately separate the data and control planes, additional support is still needed for the flexible organisation and orchestration of cooperating OpenFlow devices.

978-1-4799-0913-1/14/$31.00 © 2014 IEEE

In this paper, we present a framework that is based on policy-controlled *management patterns*. The framework provides abstractions for orchestrating individual resilience services implemented as *OpenFlow applications*, possibly over distributed controllers. A management pattern is a high-level description of the overall policy-based configuration and interactions between a set of resilience mechanisms. These include mechanisms for detection of attacks and anomalies, e.g., intrusion detection systems and bandwidth monitoring systems, and mechanisms for the remediation of these issues, e.g., traffic shaping and load balancing. A management pattern specifies how mechanisms deployed in the network must be reconfigured to address a particular type of network challenge, e.g., DDoS attack, worm propagation, large-scale disasters, etc.

The work presented here is an extension of our previous effort on defining a management framework for network resilience in the context of "classical" network deployments, i.e., those in which the data and control plane are not decoupled [1]. By demonstrating our framework's applicability to an SDN, we propose that it can be applied to resilience management in settings that include both forms of deployment model. This ability will prove important as future networks will very likely consist of resilience services that are implemented using SDN technologies and more closed devices, such as network security appliances. With respect to our previous work, this paper presents three main contributions: *(i)* to the best of our knowledge, there has been no related work that targets the issue of managing resilience services for SDNs, so we provide a significant extension in placing our work within the SDN context; *(ii)* we demonstrate how several components of the our framework can benefit from the functionality provided in an SDN, which hitherto has not been readily possible; and *(iii)* the experimental results presented in this paper, which are executed in an emulation environment, affirm and extend our previous simulation-based results [2].

This paper is organised as follows: Section II presents background on network resilience, and how OpenFlow can assist in building resilience services and applications. Section III describes our framework for the orchestration of resilience services, and how it can be applied to software-defined networks. Section IV presents initial experimental results that indicate the applicability of our framework to SDNs. Section V outlines the related work on policy abstractions for SDNs, and finally Section VI presents the concluding remarks and future work.

II. Network Resilience and Software Defined Networks

A. Network Resilience

Network resilience is the ability of a network to maintain acceptable levels of service in the face of challenges to its normal operation, such as malicious attacks, natural disasters or human errors [3]. Ensuring network resilience requires the continuous monitoring of a network's operation, trying to detect challenges that are afflicting the network and attempting their prompt mitigation. Thus a resilience strategy often requires the coordinated management of interacting detection and remediation mechanisms that operate in the infrastructure, possibly in different layers of the protocol stack and in different administrative domains.

Firstly, *detection mechanisms* support the identification and categorisation of challenges to the network. Several research efforts have focused on developing detection techniques, whose output can be used to trigger the dynamic adaptation of network resources to remediate anomalous conditions. These include signature-based systems [4], [5], [6], which are based on the detection of predetermined attack traces, as well as anomaly detection systems, which aim to identify unexpected changes in traffic volumes [7] or changes in the entropy of traffic features [8]. Configuring these systems is difficult, as different choices of sampling rate, for example, can impact their accuracy, and must be carefully fine tuned [9]. Classification techniques can be used to identify the root cause of network anomalies, and lead to a trade-off between complexity and accuracy of identification [10]. Similarly, a range of *remediation mechanisms* may be used for containing the effects of a challenge. For example, various forms of traffic shaping can be used, from simply blocking traffic to probabilistic rate limiting, which can be applied at different protocol levels and to individual network device ports. Firewalls and OpenFlow switches [11], for example, can be used to block or shape network traffic.

Thus to provide network resilience, a number of mechanisms are needed, such as monitoring systems, tools to collect IP flow information for use by intrusion detection and classification systems, and those to mitigate challenges. Despite the multitude of mechanisms and techniques available, it is often not clear how these should be combined and coordinated in complex multi-service networks. We found that the published state-of-the-art in challenge detection and classification varies in the resources that are required, the timeliness and accuracy of their operation, and the challenges they can effectively operate with [12]. For example, localised detection in fluctuations of traffic volumes can give a rapid and relatively lightweight indication of the onset of challenges, such as DDoS attacks or flash crowd events, whereas a sophisticated classification system can yield more accurate information about the challenge, e.g., the identification of malicious flows, over a longer period of time. However, there are few works investigating best practices on how to enforce network-wide resilience strategies against specific types of challenges. This must be based on the coordinated management of a subset of detection and remediation mechanisms that are suitable for a given challenge. Therefore, it must be possible to flexibly organise detection mechanisms and specify their operation in a way that is sympathetic to their characteristics and the likely challenges that will occur. Moreover, configuring these mechanisms will be complex, especially when one considers their interaction with those to remediate challenges – an issue that existing work does not address.

B. OpenFlow and Network Resilience

Software-defined networking in general, and *OpenFlow* [13] as its primary realisation, decouple the data and control planes, facilitating the configuration of routers and switches. Central to this paradigm is the OpenFlow *switch*, which has a set of internal flow tables, and a standard interface and protocol for adding and removing flow entries. A flow entry is defined by a packet header, which specifies the flow, and an action associated with it (e.g., an action might specify the dropping of packets belonging to a specific flow). OpenFlow *applications* execute on remote *controller* platforms that provide an application programming interface (API) for sending control information over a secure channel in order to, amongst other things, modify flow entries. This ability to easily modify flow entries assists network operators to program OpenFlow switches to collect statistics, and to implement networking applications.

Using this separation of control and data plane, a number of services that support resilience functionality can be readily implemented. For example, Braga *et al.* make use of OpenFlow and the NOX controller[1] to implement a traffic classification approach that can be used to detect Distributed Denial of Service (DDoS) attacks using self-organising maps [14]. In this case, OpenFlow is used to collect flow statistics that can be used as part of the classification process. Within the SDN community there is ongoing work on defining high-level languages and frameworks for the specification of network policies, which are realised using OpenFlow [15] – these can be used for resilience purposes. For example, in the Pyretic [16] software distribution, a sample implementation of a firewall is given[2].

Policy languages, such as Pyretic, can be used to rapidly implement resilience functionality. An example is shown in Fig. 1, whereby a policy specifies that the number of packets should be counted, which are destined for a Web service listening on port 80 at the IP address 10.0.0.1, per source IP address in five second intervals. A callback is registered for another policy that specifies if more than 200 packets have been counted, using the aforementioned policy, the packets for the source IP address associated with the counter and the Web server address should be dropped. In the Pyretic

[1]http://www.noxrepo.org/

[2]http://www.frenetic-lang.org/pyretic/

```
class monitor_webserver(DynamicPolicy):
 def main(self):
  Q = count_packets(interval=5, group_by=['srcip'])
  match(IP('10.0.0.1')) & match(dstport=80) >> Q
  Q.register_callback(self.drop_malicious)

 def drop_malicious(count,pkt):
  if (count > 200):
        match(srcip=IP(pkt['srcip']))
      & match(dstip=IP('10.0.0.1')) >> drop
```

Fig. 1. An example Pyretic policy that counts the number of packets to a Web server and drops a host's traffic if more than 200 are sent in five seconds

framework, this simple high-level policy specification executes in the Pyretic run-time, which interfaces with a small OpenFlow application that realises the underlying interactions with an OpenFlow switch. We assume that a range of resilience services, such as link monitoring and traffic shaping, can be implemented as OpenFlow applications in a similar manner, using either Pyretic or other high-level policy frameworks, such as Procera [17].

Despite the existence of such policy frameworks, their coordination across multiple instances that implement resilience functionality is largely left unattended. This is highlighted by the important distinction between the *southbound* and the *northbound* interfaces proposed in SDN management [18]. The southbound interface refers to interaction between SDN switches and the controller, e.g., the OpenFlow protocol. The northbound interface concerns the representation of network-wide policies and how these are translated into the configuration of the controller. This is illustrated in Fig. 2. Although there has been considerable research effort in defining the southbound interface, there is comparatively little development so far with respect to the northbound interface [18]. In particular, it is not clear how a range of resilience services, which could be realised in a number of ways, e.g., using frameworks like Pyretic, implemented as "low-level" OpenFlow applications using platforms such as NOX, or as "classical" tightly-coupled services, can be consistently orchestrated and managed. Building on our previous work, here we show how our resilience management framework can be applied in a software-defined network, and can therefore be consistently used across these deployment approaches to orchestrate and manage resilience strategies.

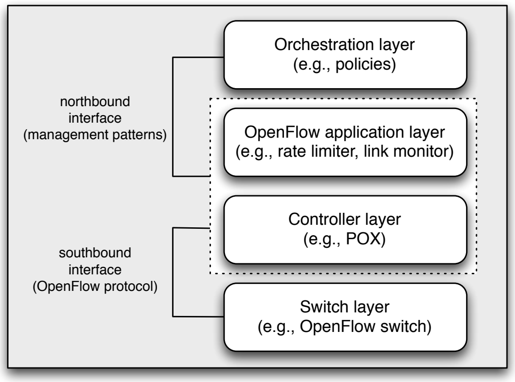

Fig. 2. Layers in SDN management.

III. Resilience Management for SDNs

In previous work [1], we have developed a resilience management framework, which supports a number of functions including the specification and evaluation of resilience strategies, the on-line measurement and evaluation of the resilience of a network, and the deployment of strategies on a network. The management framework is summarised in Fig. 3. At the centre of the framework is a *resilience manager* that supports real-time policy-driven adaptation of managed resilience services, such as intrusion detection systems and firewalls, in order to address challenges. Providing input to the resilience manager is a *challenge analysis* component, whose purpose is to develop situational awareness about the existence and nature of ongoing challenges. In what follows, we summarise the resilience management functions that are highlighted in Fig. 3, and discuss how they can be applied in the context of SDNs, including the specification of resilience strategies.

A. Resilience Strategy Specification

To assist the specification of resilience strategies, our framework relies on the notion of *management patterns* [1]. A management pattern is a high-level specification template that describes the overall policy-based configuration and interactions between a set of resilience mechanisms. A management pattern specifies how mechanisms deployed in the network infrastructure, implemented, e.g., as OpenFlow applications, must be reconfigured in order to address a particular type of network challenge. Management patterns are akin to software design patterns in the sense that patterns promote the use of well-established solutions to recurring problems. Thus, a management pattern provides reusable solutions to recurring resilience problems, but also prescribes policy templates to address them using a set of resilience mechanisms. Broadly, resilience mechanisms are organised in two categories: they can either implement mechanisms for *detection* of network challenges, or they can be used for *remediation* of those challenges. Detection mechanisms are event sources, which include intrusion detection systems and network bandwidth monitoring systems, for example. Remediation mechanisms can be used for reconfiguring the network in response to monitored events, such as traffic shaping or load balancing.

Management patterns are specified in terms of *roles*, to which management functions and policies are associated. Roles can be used to represent abstractly OpenFlow applications (e.g., rate limiter, link monitor, traffic classifier, etc.) that implement some kind of resilience functionality. A management pattern is specified off-line, based on the functionality expected from a set of roles. But during run-time, OpenFlow application instances will be assigned to these roles, based on the availability of the devices associated with a given network. For example, a pattern for combating a flash crowd may include roles such as *VirtualMachineReplicator* and *WebServerMonitor*, whereas a pattern for addressing a DDoS attack may include roles such as *TrafficClassifier* and *RateLimiter*.

Management patterns make use of previous work for systematically constructing interactions between policy-based systems in terms of three complementary perspectives [19]:

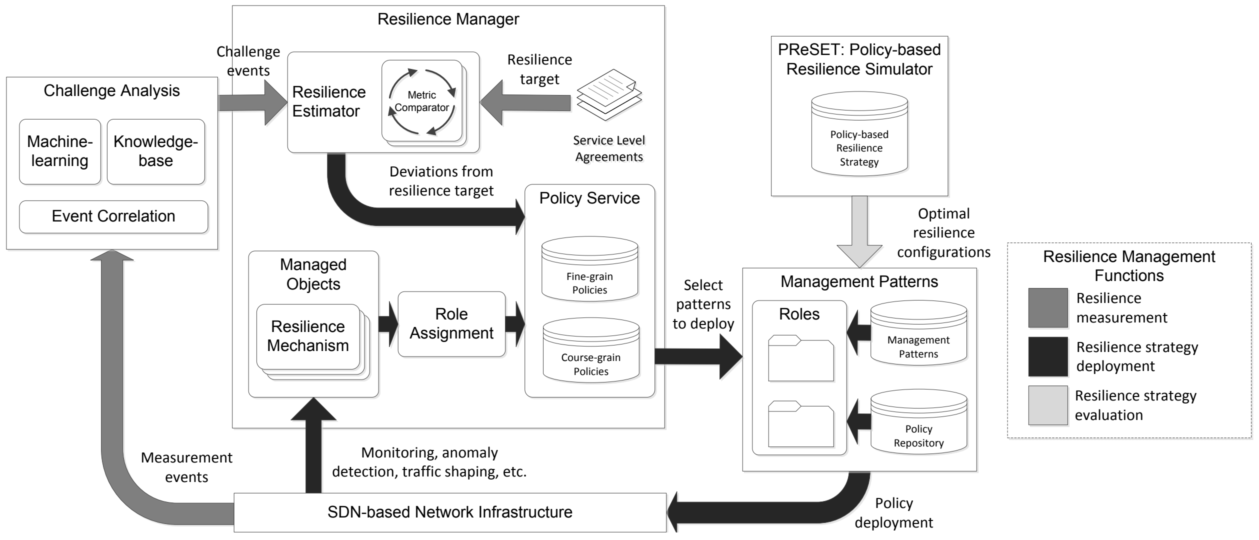

Fig. 3. Resilience management framework, showing management functions

- ***Communication (event exchanges)***: events are necessary for triggering *event-condition-action* ECA policies. Event exchanges can be enforced, for example, through a *blackboard* communication model, a simple *diffusion* between a source and a target, or support some form of event *correlation* to produce a high-level event.
- ***Task-allocation (policy exchanges)***: dynamically loading new policies into a device enables the easy reconfiguration and adaptation of its behaviour during runtime. Task-allocation via policy exchanges can be based on a *hierarchical control* or a more elaborate *bidding* approach between groups of devices, for example.
- ***Structure (interface exchanges)***: interfaces are necessary for validating the actions prescribed by a policy. Access to interfaces can also be achieved in different ways, for example, via an encapsulated *composition*, a *peer-to-peer* interaction, or a multi-level service *aggregation*, which is not necessarily encapsulated.

To encode different abstractions for achieving event, policy and interface exchanges we can express them in terms of management relationships. A catalogue of common management relationships, such as *p2p*, *event diffusion*, and *hierarchical control*, has been presented in [19]. Each defines its own specific management functions (e.g., *source/target*, for *event diffusion*). These primitive management relationships can be seen as building blocks, which are associated with two or more roles in a pattern. This enables sophisticated event sharing and alert forwarding strategies to be systematically built out of simple primitives. Event sources can be defined and event sharing schemes can be implement using high-level management relationships. Similarly, event-driven policies and policy loading strategies can be specified using these primitives. A management pattern combines the behaviour of a set of management relationships to prescribe the behaviour of the roles within the pattern.

We advocate the use of management patterns to realise the specification of the *northbound interface* in Fig. 2. Management patterns can encode the policy configurations between OpenFlow applications, and specify how event sources share information with the other components in a resilience strategy. Patterns are implemented in the Ponder2 policy framework[3]. The configuration policies in a management pattern are standard Ponder2 policies. Event forwarding, interface exchange and policy loading protocols are also implemented in Ponder2. In previous work [1], we describe in details the specification of patterns, including its parameterisation with roles and the establishment of management relationships between them.

Further, high-level management patterns are translated into low-level device configuration. The OpenFlow application receives a reconfiguration request based on its role and the management functions associated with the role. This is translated to packet forwarding rules and flow table reconfigurations (to insert, delete, or modify rules) that will be transmitted to the OpenFlow-enabled switch using the OpenFlow protocol.

B. Resilience Measurement

An important resilience management function is to measure the current resilience status of the network, and determine when it is not at desired levels. In our resilience management framework, deviations from a resilience target, expressed in a Service Level Agreement (SLA), can lead to fine-grain adaptation of the infrastructure, such as the adjustment of parameters associated with resilience services. In cases when there are significant deviations, coarse-grain adaptation, i.e., the invocation of new management patterns, can occur. Central to this process is the *resilience estimator*, which takes as input measurement information and a resilience target expressed in an SLA, and generates an event when deviations occur.

Previous work has suggested that measuring the resilience of networks should be undertaken at multiple levels of the network protocol stack [20], [21]. With this in mind, SDNs can support the implementation of resilience measurement in

[3]http://www.ponder2.net

a number of ways. Topological metrics, e.g., average node degree and betweenness, can provide useful insights into the resilience of a network. Arguably, by making use of a logically centralised control functionality, an overview of the network topology can be more readily achieved. For instance, this information can be made available as a by-product of control functionality that implements routing.

Furthermore, specific resilience measurement functionality, implemented as OpenFlow applications, can be deployed on OpenFlow controllers in order to measure the items expressed in an SLA. These measurement applications can take advantage of OpenFlow to collect useful *counter* data available at switches. For example, data, such as the received and transmitted packets or bytes associated with a flow or port, can give indications of the current performability[4] of the network. Additionally, OpenFlow counters are available that relate to receiving and transmission errors, which may indicate the onset of challenges. A summary of these is shown in Table I.

Per Port	*Per Queue*
Receive & Transmit Drops	Transmit Overrun Errors
Receive & Transmit Errors	
Receive Frame Alignment Errors	
Receive Overrun Errors	
Receive CRC Errors	

TABLE I. OPENFLOW ERROR COUNTERS

In our framework, the *resilience estimator* itself can be viewed as an OpenFlow application, which makes use of the aforementioned counters and other information, such as the current network topology, in order to invoke adaptations that address resilience issues.

C. Resilience Strategy Deployment

In our framework, a *challenge analysis* module collects events from the network infrastructure, such as the output from intrusion detection systems, so they can be correlated to enable the identification of a challenge. There are a number of techniques that can be used for correlation. *Statistical* correlation techniques use statistical features of events, and do not require previous knowledge of an attack to function [22]. Correspondingly, they can be used to correlate previously unseen attacks. Meanwhile, correlation techniques, which identify *similarities* between events largely use clustering techniques to group them based on their features [23]. A general drawback of these approaches is that empirically acquired domain knowledge is required to configure the clustering mechanisms, although some exceptions exist [24]. There are techniques that correlate events through the use of a *knowledge-base*, for example, that describe an attack scenario [25]. The major difficulty and overhead of using knowledge-based approaches is creating the models used for correlation. We advocate the use of a hybrid approach that makes use of a knowledge-base for well-understood challenges, such as those examined in the PReSET tool (see Section III-D), and capitalise on machine learning-based approaches to correlate previously unseen behaviours.

When the challenge analysis module identifies a challenge, *coarse-grain ECA policies* determine which patterns must be deployed. As previously mentioned, in our management framework, resilience services that exist in the network, e.g., on firewalls, routers and switches, are assigned to roles, such as *link monitor* and *rate limiter*, that are specified in management patterns. In a "classical" network, i.e., not an SDN, resilience services are tightly coupled with the devices that implement them. Assignment of roles to resilience services involves the task of associating services, and their implementation on a device, to a specific role. In a software-defined network, role assignment involves associating an OpenFlow application and an associated set of switches, which realise a resilience service, to a role (Fig. 4). We assume that, in an SDN, many different OpenFlow applications could be eligible for enforcing a resilience strategy and its policies.

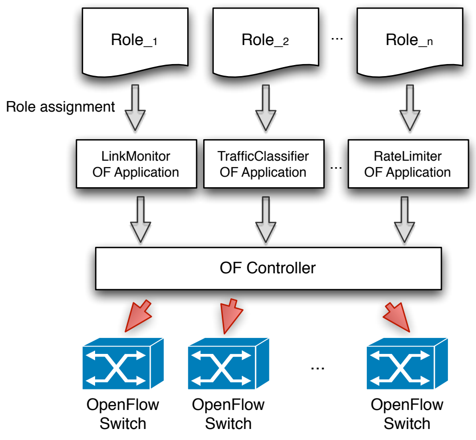

Fig. 4. Role assignment performs the matching of the requirements of a role to the functionality offered by an OpenFlow application.

Role assignment is achieved through matching the requirements of the roles specified in the pattern (in terms of the policies that a service must be capable of enforcing or the events that it must be capable of handling) to the interface of an OpenFlow application. More specifically, role assignment is defined in terms of a set of *events (E)* that must be sent, *notifications (N)* that must be handled, and *operations (O)* that must be supported by an OpenfFlow application. Formally, let $Itf_m = \langle O_m, E_m, N_m \rangle$ be the management interface of an application, let r be a role, and let $Req_r = \langle O_r, E_r, N_r \rangle$ be the requirements for that role, then the assignment of the OF application which has Itf_m to role r is subject to the following:

$$assign(Itf_m, r) \rightarrow (O_r \subseteq O_m) \wedge (E_r \subseteq E_m) \wedge (N_r \subseteq N_m)$$

Role assignment is not concerned with the mapping and physical distribution of OpenFlow controllers in the network topology [26], [27], or the embedding of virtual networks into a given physical network substrate [28]. Instead, role assignment can be seen as an optimisation problem of finding the best subset of resources *"capable of enforcing a specific management pattern and its policies"*. In addition to requiring OF applications that match the roles requirements, role assignment may also take into account a number of pre-specified constraints such as CPU, memory and bandwidth requirements, or location. As we assume that many different OF applications could be eligible for enforcing a resilience

[4]"Performability is that property of a computer system such that it delivers performance required by the service, as described by QoS (quality of service) measures."[3]

strategy, we thus aim to investigate how the selection and association of different applications to roles can be performed in an optimal manner, considering the several alternatives. Currently, role assignment is performed manually, but as future work we intend to formulate it as an optimisation problem, using a *mixed integer programming* (MIP) [29] formulation. Further, we aim to support role assignment in a flexible manner, catering for reassignment according to the resources available, and on-demand.

Upon role assignment, event and policy exchanges between OpenFlow applications occur according to the pattern specification. For example, a pattern specification may use a *task-loading management relationship* to define that the following policy should be sent to reconfigure the application running the *RateLimiter* role:

```
policy throttling {          //loaded into RateLimiter
  on notify_detection(IPAddress, link)
      do limit(IPAddress, link, %x);
}
```

The policy above specifies how a naïve yet preventive traffic shaping should be applied to all traffic destined to a specific IP address and passing through a specific link, if an anomaly event is received. When receiving this policy, the rate limiter application performs its reconfiguration, ultimately communicating with the controller and updating the flow table with a new rule on the corresponding OpenFlow switch. For instance, the *limit* action can be realised by introducing a flow rule that maps flows that are destined for *IPAddress* to a pre-configured *queue* associated with a *port*, i.e., a *link*. An OpenFlow switch can be configured with a number of queues on its ports, and the ports be assigned a minimum and maximum rate, specified as a percentage – equivalent to the ***%x*** parameter in the aforementioned policy. Flows that are mapped to a specific queue are treated according to that queues configuration; in this case, subjected to a maximum rate.

Similarly, a pattern specification may also define the necessary event forwardings between OpenFlow applications. In particular, the snippet below defines that the application running the *AnomalyDetection* role should forward *notify_detection* events to the application running the *RateLimiter* role using a *diffusion management relationship*. This instructs the OpenFlow application running the *AnomalyDetection* role to collect specific flow statistics in the respective OpenFlow switch and forward a corresponding event to the rate limiting application.

```
diffusion (target RateLimiter,
           source AnomalyDetection)
               event: notify_detection(IPAddress, link);
```

A pattern defines the overall configuration of mechanisms, but the parameters and thresholds of these mechanisms are defined according to *fine-grain ECA policies*, which take as input the most recent indication of the state of the network, as published by the resilience estimator. To accommodate this form of policy-driven adaptation, OpenFlow applications that implement resilience functionality must expose an interface that can be invoked by the Ponder2 framework.

D. Resilience Strategy Evaluation

In previous work [2], we have developed a tool called PReSET for simulating resilience strategies, which is based on a coupling of the OMNET++ network simulator and the Ponder2 policy-based management framework. The purpose of this tool is to evaluate the effectiveness of resilience strategies that address a particular challenge, which can subsequently be promoted to resilience management patterns. Such a tool as PReSET is necessary because validating resilience strategies on live networks is often not possible, and testbed facilities are typically limited in scale; this is in contrast to the nature of a number of pertinent challenges such as DDoS attacks, Internet worms and botnets, which involve large numbers of topologically distributed hosts that generate significant volumes of traffic. In PReSET, resilience mechanisms are implemented as OMNeT++ modules that can generate events that are sent to an external Ponder2 instance, and have actions invoked upon them based on policies. This approach enables policies that are shown to perform well via simulation to be readily implemented on a real network deployment with little additional effort.

Because of the design of PReSET, integrating resilience strategies that make use of OpenFlow functionality is relatively straightforward. Klein and Jarschel [30] have implemented OMNeT++ modules for OpenFlow, including modules that model an OpenFlow switch, controller and a number of simple applications, such as a hub and a switch. Integration of these modules into the PReSET tool can be readily achieved. Resilience services that are realised as *OF_Controller_App* modules – a dummy OpenFlow application that communicates with an *OF_Controller* module to receive packet-in signals – can be extended to interface with the Ponder2 framework. To do this requires the implementation of a simple XMLRPC server within the simulation that enables communication to the external Ponder instance. Further details about how to achieve this integration can be found in [2]. At the time of writing, a shortcoming of the OpenFlow model implemented by Klein and Jarschel is that a number of OpenFlow messages are not supported, including the `OFPT_STATS_REQUEST` message, which can be used to collect important statistics for resilience measurement, as discussed in Sec. III-B. This is clearly an area for future work.

IV. Experimental Results

To demonstrate the applicability of our resilience management framework to SDNs, we have carried out initial experiments. We have used the *Mininet*[5] framework to create a virtual network, based on Open vSwitch[6]. Mininet utilizes network namespaces, a feature of the Linux kernel, to implement lightweight network virtualization. Nodes in a Mininet network, which represent hosts, switches and controllers, can execute arbitrary applications that are available on the file system. Additionally, switches that are modelled in Mininet can connect to remote OpenFlow controllers; we made use of this facility for our experiments.

To realise an exemplar resilience service, we created a POX-based[7] OpenFlow application (module), whose purpose is to monitor Web traffic at a switch and drop traffic from aggressive hosts. To do this, the POX module periodically

[5]Mininet: http://mininet.org/
[6]Open vSwitch: http://openvswitch.org/
[7]POX: http://www.noxrepo.org/pox/about-pox/

collects flow statistics from a switch and calculates the amount of traffic in bytes destined for TCP port 80 (Web traffic), for each source data link layer address it observes. At each *period*, the module checks if the bytes sent for each host has exceeded a *transmission threshold*. If this threshold has been exceeded, the host acquires a penalty. A host can incur a *penalty threshold* number of penalties, after which further flows from that host are dropped by the switch. The aforementioned parameters are configured via policies, in a similar manner to the examples discussed earlier in the paper. We envisage the resilience service being invoked via policies if unusual load is observed at a Web server, for example, thus addressing a potential problem in the network. A code sample, illustrating how these parameters and the modules core behaviour can be implemented in POX, is shown in Fig. 5.

```
# Iterate through a dictionary that holds the byte count
# per source address collected during this period
for dl_src in counts.keys():
 # Check if the host has exceeded the transmission_threhold
 if counts[dl_src] > transmission_threshold:
   if penalties.has_key(dl_src):
    # Increase the penalty for this host
    penalties[dl_src] += 1
    # Check if the host has exceeded its penalty_threshold
    if penalties[dl_src] > penalty_threshold:
      for connection in core.openflow._connections.values():
          # Send OpenFlow message for host with
          # non-specified action, causing dropping
          msg = of.ofp_flow_mod()
          my_match = of.ofp_match()
          my_match.dl_src = EthAddr(dl_src)
          msg.match = my_match
          msg.priority = 65535
          connection.send(msg)
   else:
    penalties[dl_src] = 1
```

Fig. 5. A POX code snippet that can be used to implement a resilience service that drops traffic from hosts generating aggresive quantities of traffic

To test our OpenFlow-based resilience service, we created a network, using the Mininet framework, which consisted of a number of hosts connected to a single OpenFlow switch. The switch was configured to connect to a remote POX-based controller, running the *forwarding.l2_learning* module, which implements a basic learning switch, and our resilience module. A Web server was started on a host in the network, and "background" traffic was generated by hosts using the *wget* application. At a specified time, a host "attacked" the Web server, implemented using the Apache Webserver benchmarking tool, called *ab*. For analysis, traffic was captured on the host running the Web server using *tcpdump*. This behaviour was implemented in a Python script, which made use of the Mininet API.

The results of our experiment are shown in Fig. 6. Initially, background traffic is generated; at 20 secs the attack is launched at the Web server. Our resilience module was configured with a *penalty_threshold* of four. After incurring four penalties, a flow rule, which causes packets to be dropped for the specified flow, is installed on the switch for the host conducting the attack. This behaviour can be seen shortly after 40 secs into the experiment, whereby traffic levels observed at the Web server return to normal levels. (The "sawing" behaviour during the attack period, which can be seen in Fig. 6, is caused by the *ab* application stopping and restarting on the attacking host.) These preliminary experimental results

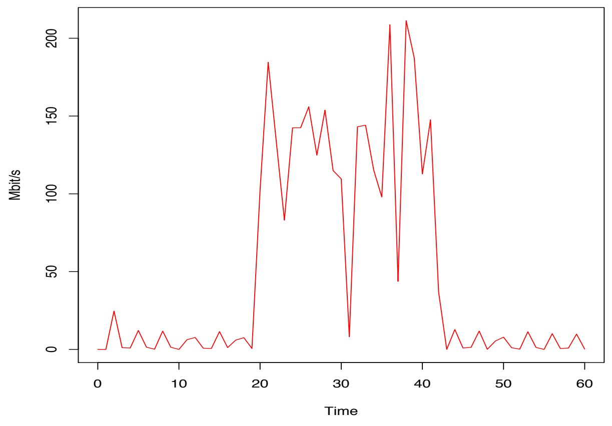

Fig. 6. Traffic measured at a Web server; an attack is launched at 20 secs and mitigated by dropping associated flows at 40 secs

demonstrate the potential to implement resilience functionality using OpenFlow and have this functionality orchestrated using the resilience management framework, which is discussed in the paper. Further work will involve elaborating on this scenario, in order to incorporate additional resilience services that can be organised into strategies, specified as patterns, and configured using further policies.

V. Related Work

SDN policy abstractions are now widely recognised as an important element in building layered, modular applications. As a result, several research efforts have defined high-level languages for specifying SDN applications. In particular, policy abstractions have been suggested for assisting the building of OpenFlow applications. Although OpenFlow provides a standard interface for manipulating rules on the switches, the controller programming model still presents a low-level programming interface that is error prone. Policy abstractions are capable of hiding the low-level configuration details (e.g., bit patterns for specifying sets of packets and integer priorities for disambiguating rules), and only present relevant information to the upper layers.

Frenetic [31] provides a high-level, declarative query language for aggregating network traffic, as well as for specifying high-level packet forwarding policies. For example, Frenetic supports the specification of policies using expressive *filter patterns*, such as `intersection`, `union`, `difference` and `complement`. It also supports the *parallel composition* of policies, which gives each module (e.g. routing and monitoring) the perception of operating on its own copy of each packet, thereby supporting code reuse. The runtime abstracts the details related to translating these policies to switch-level rules, and installing/uninstalling rules on the switches.

Pyretic [16] is a language and system that enables programmers to specify policies at a high-level of abstraction. It extends the work developed in Frenetic, and introduces the *sequential composition* of policies, which allows a module to act on packets already processed by another module (e.g. routing after load balancing). Pyretic also supports abstract network topologies, in which each module applies a policy over its own abstract view of the topology, thereby supporting information hiding and protection.

Procera [17] is a declarative policy language for SDNs based on functional reactive programming. It provides a collection of domain-specific operators for expressing temporal queries, defined over external events such as bandwidth use, intrusion detection or specific time events. Procera provides an event algebra that supports operations for filtering, transforming and merging event streams. The policy layer sits on top of the controller and can react to signals about network events as well as out-of-band signals from users and external devices. Procera is extensible so users can extend the language by adding new constructs and event operators.

Our work is complementary to the policy languages described above. We assume that OpenFlow applications will still be built using high-level languages like Frenetic, Pyretic and Procera. However, our focus is not on building individual applications, but instead on how to combine and orchestrate these individual applications, possibly implemented over distributed controllers, into network-wide resilience services. We expect that management patterns can assist in providing these much need abstractions.

VI. Concluding Remarks

Software-defined networking provides a flexible architecture for the configuration of network devices, based on the decoupling of the data and control planes. Although software-defined networking in general, and OpenFlow in particular, provide such abstractions, support is still needed for orchestrating a collection of devices and services that must cooperate to implement network-wide resilience services. To address this issue, we advocate the use of *management patterns* as abstractions for orchestrating individual resilience services implemented as OpenFlow applications, possibly over distributed controllers.

In the framework presented in this paper, management patterns can be used to specify the management relationships between resilience mechanism types to combat specific types of network challenges. Based on the mechanisms required by each pattern, OpenFlow applications deployed in the network are chosen and configured to perform event exchanges and policy reconfigurations as prescribed by the overall pattern. The benefits of using management patterns are two-fold: *(1)* they can be used to systematically build network-wide resilience services using building-block abstractions that define common management relationships; *(2)* patterns are abstract specifications for containing specific types of challenges, so they can be reused in different parts of the network or with the devices currently available in a given network.

As part of our immediate future work, we are going to expand on our experimental results using the Mininet emulator. We intend to emulate a wider range of network challenges and attacks, and perform a thorough assessment of network reconfigurations using management patterns. We also intend to work on the integration of the PReSET [2] toolset for resilience simulation with an OMNeT++ implementation of OpenFlow [30]. We expect to be able to evaluate policy-based resilience strategies based on OpenFlow reconfigurations also using the simulation environment.

Acknowledgements

The research presented in the paper was supported by the EU-funded PRECYSE (www.precyse.eu) and SECCRIT (www.seccrit.eu) projects, with contract numbers 285181 and 312758, respectively.

References

[1] A. Schaeffer-Filho, P. Smith, A. Mauthe, D. Hutchison, Y. Yu, and M. Fry, “A framework for the design and evaluation of network resilience management,” in *Proceedings of the 13th IEEE/IFIP Network Operations and Management Symposium (NOMS 2012)*. Maui, Hawaii, USA: IEEE Computer Society, April 2012, pp. 401–408.

[2] A. Schaeffer-Filho, A. Mauthe, D. Hutchison, P. Smith, Y. Yu, and M. Fry, “PReSET: A toolset for the evaluation of network resilience strategies,” in *Proceedings of the IFIP/IEEE Integrated Network Management Symposium (IM 2013)*. Ghent, Belgium: IEEE Computer Society, May 2013, pp. 202–209.

[3] J. P. G. Sterbenz, D. Hutchison, E. K. Çetinkaya, A. Jabbar, J. P. Rohrer, M. Schöller, and P. Smith, “Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines,” *Computer Networks: Special Issue on Resilient and Survivable Networks (COMNET)*, vol. 54, no. 8, pp. 1245–1265, June 2010.

[4] M. Roesch, “Snort - lightweight intrusion detection for networks,” in *Proceedings of the 13th USENIX conference on System administration*. Berkeley, CA, USA: USENIX Association, 1999, pp. 229–238.

[5] Cisco, “Cisco intrusion prevention system,” Available at: http://www.cisco.com/go/ips. Accessed in: February 2011.

[6] IBM, “Security network intrusion prevention system,” Available at: http://www-01.ibm.com/software/tivoli/products/security-network-intrusion-prevention/. Accessed in: February 2011.

[7] A. Hussain, J. Heidemann, and C. Papadopoulos, “A framework for classifying denial of service attacks,” in *SIGCOMM ’03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*. New York, NY, USA: ACM, 2003, pp. 99–110.

[8] A. Lakhina, M. Crovella, and C. Diot, “Mining anomalies using traffic feature distributions,” *SIGCOMM Comput. Commun. Rev.*, vol. 35, no. 4, pp. 217–228, 2005.

[9] D. Brauckhoff, K. Salamatian, and M. May, “A signal processing view on packet sampling and anomaly detection,” in *INFOCOM ’10: Proceedings of the 29th Conference on Information Communications*. Piscataway, NJ, USA: IEEE Press, 2010, pp. 713–721.

[10] T. Peng, C. Leckie, and K. Ramamohanarao, “Survey of network-based defense mechanisms countering the DoS and DDoS problems,” *ACM Comput. Surv.*, vol. 39, no. 1, p. 3, 2007.

[11] T. A. Limoncelli, “OpenFlow: a radical new idea in networking,” *Commun. ACM*, vol. 55, no. 8, pp. 42–47, Aug. 2012.

[12] Y. Yu, M. Fry, A. Schaeffer-Filho, P. Smith, and D. Hutchison, “An adaptive approach to network resilience: Evolving challenge detection and mitigation,” in *DRCN’11: 8th International Workshop on Design of Reliable Communication Networks*, Krakow, Poland, October 2011, pp. 172 –179.

[13] 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.

[14] R. Braga *et al.*, “Lightweight DDoS flooding attack detection using NOX/OpenFlow,” in *35th IEEE Conference on Local Computer Networks (LCN)*, Denver, Colorado, USA, October 2010.

[15] N. Foster, A. Guha, M. Reitblatt, A. Story, M. Freedman, N. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, D. Walker, and R. Harrison, “Languages for software-defined networks,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 128–134, 2013.

[16] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker, “Composing software-defined networks,” in *Proceedings of the 10th USENIX conference on Networked Systems Design and Implementation*. Berkeley, CA, USA: USENIX Association, 2013, pp. 1–14.

[17] A. Voellmy, H. Kim, and N. Feamster, “Procera: a language for high-level reactive network control,” in *First workshop on Hot topics in software defined networks*, ser. HotSDN ’12. New York, NY, USA: ACM, 2012, pp. 43–48.

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

[19] A. Schaeffer-Filho, “Supporting management interaction and composition of self-managed cells,” Ph.D. dissertation, Imperial College London, 2009.

[20] E. K. Çetinkaya, M. J. Alenazi, A. M. Peck, J. P. Rohrer, and J. P. G. Sterbenz, “Multilevel Resilience Analysis of Transportation and Communication Networks,” *Springer Telecommunication Systems Journal*, 2013, accepted on July 2013.

[21] C. Doerr and J. Hernandez, “A Computational Approach to Multi-level Analysis of Network Resilience,” in *Third International Conference on Dependability (DEPEND), 2010*, 2010, pp. 125–132.

[22] W. Lee and X. Qin, “Statistical Causality Analysis of Infosec Alert Data,” in *Managing Cyber Threats*, ser. Massive Computing, V. Kumar, J. Srivastava, and A. Lazarevic, Eds. Springer, 2005, vol. 5, pp. 101–127.

[23] S. Lee, B. Chung, H. Kim, Y. Lee, C. Park, and H. Yoon, “Real-time analysis of intrusion detection alerts via correlation,” *Computers & Security*, vol. 25, no. 3, pp. 169–183, May 2006.

[24] J. Song, H. Takakura, Y. Okabe, and K. Nakao, “Toward a more practical unsupervised anomaly detection system,” *Information Sciences*, vol. 231, pp. 4–14, 2013.

[25] D. Yu and D. Frincke, “Improving the quality of alerts and predicting intruder’s next goal with Hidden Colored Petri-Net,” *Computer Networks*, vol. 51, no. 3, pp. 632–654, February 2007.

[26] S. Yeganeh, A. Tootoonchian, and Y. Ganjali, “On scalability of software-defined networking,” *Communications Magazine, IEEE*, vol. 51, no. 2, pp. 136–141, 2013.

[27] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem,” in *Proceedings of the first workshop on Hot topics in software defined networks*. New York, NY, USA: ACM, 2012, pp. 7–12.

[28] X. Cheng, S. Su, Z. Zhang, K. Shuang, F. Yang, Y. Luo, and J. Wang, “Virtual network embedding through topology awareness and optimization,” *Computer Networks*, vol. 56, no. 6, pp. 1797–1813, 2012.

[29] A. J. Osiadacz, “Integer and combinatorial optimization,” *International Journal of Adaptive Control and Signal Processing*, vol. 4, no. 4, pp. 333–334, 1990.

[30] D. Klein and M. Jarschel, “An OpenFlow Extension for the OMNeT++ INET Framework,” in *6th International Workshop on OMNeT++*, Cannes, France, March 2013.

[31] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: a network programming language,” in *16th ACM SIGPLAN international conference on Functional programming*. New York, NY, USA: ACM, 2011, pp. 279–291.