SDN Middlebox Architecture for Resilient Transfers

Pradeeban Kathiravelu
INESC-ID Lisboa
Instituto Superior Técnico
Universidade de Lisboa, Portugal
pradeeban.kathiravelu@tecnico.ulisboa.pt

Luís Veiga
INESC-ID Lisboa
Instituto Superior Técnico
Universidade de Lisboa, Portugal
luis.veiga@inesc-id.pt

***Abstract*—Leveraging Software-Defined Networking (SDN) and middleboxes, application-level policies can be propagated to the network. *SMART* is an SDN middlebox architecture that differentiates network flows based on tenant inputs. By leveraging FlowTags software middlebox in addition to the OpenFlow rules, it supports a larger scope of tenant preferences and rules from the application layer to alter the network flow behaviour. It thus ensures timely delivery of priority flows by dynamically diverting them to a less congested path or even cloning the packets of higher priority flows along with the original flow.**

I. Introduction

Enterprise tenant network flows have various constraints to meet. *SMART* is an architectural enhancement aiming to offer a resilient transfer for critical flows, based on tenant preferences that are passed to the network layer from the processes or applications executing in the servers. *SMART* leverages redundancy through SDN and FlowTags [1] middlebox architecture to ensure timely delivery of critical flows following: i) a **divert approach** that changes the course of the latter subflow of flows in an alternative direction towards the destination in case of a detected congestion or a network failure in the middle of a transfer, typically identified through a perceived delay in flow completion time as opposed to an estimate or a specified deadline/threshold; or in the same situation for higher priority flows ii) a **clone approach** that creates a duplicate of the latter subflow and routes it in an alternative path along with the original flow unmodified.

Figure 1 depicts the *SMART* deployment, separated into a i) control plane consisting of the FlowTags-capable SDN controller and *SMART* components, and a ii) data plane consisting of the nodes - servers/hosts and OpenFlow-capable switches. The switches construct the network by connecting each other as well as the servers where the flows originate. Originally a POX extension, we redesign the FlowTags controller as an OSGi bundle to be deployed in Apache Karaf container of OpenDaylight. This modularized architecture enables plugging in of *SMART* components in control plane.

FlowTagger and ***Rules Manager*** are *SMART* components that are developed as FlowTags-capable software middleboxes. While in a typical FlowTags deployment an existing middlebox such as an intrusion detection system (IDS) is extended to read and write the tags, these *SMART* components have no specific functionality other than handling the tags and communicating with the FlowTags controller by invoking its API to generate or consume the tags in a unified manner, as presented by the FlowTags architecture [1].

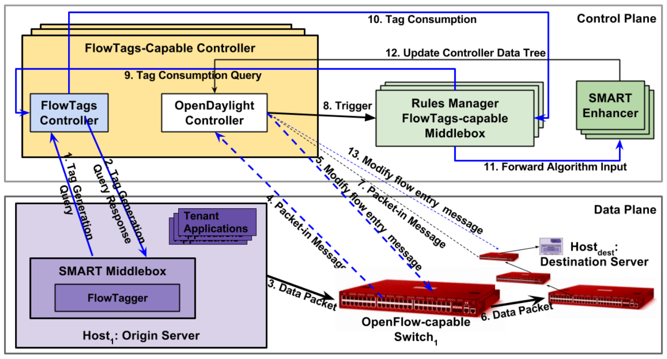

Fig. 1. *SMART* Deployment

FlowTagger is a generator/writer of the tags, similar to the FlowTags-capable middleboxes for NATs. It is deployed in each of the hosts. Packets of a selected subset of flows, defined as the priority or critical flows according to the tenant applications, are tagged while the other flows are left unmodified. Hence, the entire *SMART* enhancement workflow is initialized only on the priority flows as identified from the application layer. Following the packet processing walk-through for tag generation, FlowTagger initially sends the tag generation query to the FlowTags controller, and receives the tag generation query response. FlowTagger modifies the packet headers accordingly, and the data packets are transferred in its original path through the switches.

OpenFlow switches in the flow path communicate with the OpenDaylight controller through the OpenFlow API. A packet-in message is sent by the switches to the controller, and in turn the switches receive the modify flow entry message from the controller. The data flows continue through the intermediary nodes/switches. Later, at a policy violation, as identified from the tagged packets of the priority flows, the controller is invoked again through the OpenFlow API. This further triggers the Rules Manager in the control plane.

Rules Manager is a consumer of tags, similar to the FlowTags-capable firewalls that reads and interprets the tags. As only the first packets of the violating flows are sent to the control plane, control plane becomes the ideal location to retrieve the tags from the controller and read them by the Rules Manager, rather than as an additional middlebox in the data plane. Rules Manager sends the tag consumption query and receives the tag consumption response from the FlowTags API. It further forwards the contextual information of the packets of the flow in question to the **SMART *Enhancer***, which is responsible for computing the routing decisions and propagating them to the SDN controller. Thus, the OpenDaylight's data tree, the data structure storing distributed objects inside the controller, is updated by the Enhancer. Thus the controller updates the flow tables based on the Enhancer output.

II. *SMART* Redundancy in Network Flows

A set of algorithms has been developed as part of *SMART* Enhancer that is deployed along with the extended FlowTags-capable OpenDaylight SDN controller. $\mathit{SMARTRoute}$, the core routing procedure is described in Algorithm 1. For the ease of expression, Algorithm 1 (and the rest of this section) assumes clone approach to be the default, while referring to both clone and divert approaches. The latter subsets of packets of priority flows, known as subflows, are diverted/cloned when the current routing fails to complete the transmission of the flow within the stipulated soft limit. These limits, set by the controller on the switches, will trigger a communication to the controller from the switches when a violation of a hard limit or threshold is imminent. Soft limit parameters are often modelled as a fraction of the respective hard limit parameters such as flow completion time. Tags such as priority and SLA parameters are added to the packets of the flows to provide the additional information required in accomplishing this.

Algorithm 1 *SMART* Enhancement

```
1: procedure SMARTRoute($flow$, $origin$, $dest$)
2:   repeat
3:     $BaseRoutingAlgorithm(flow, origin, dest)$
4:   if ($flow.policies.isThresholdMet()$) then
5:     $cloneOrigin \leftarrow markBreakPoint(flow, origin, dest)$
6:     $cloneDest \leftarrow findCloneDest(flow, flow.status)$
7:     $clonedFlow \leftarrow cloneFlow(flow, cloneOrigin, cloneDest)$
8:     $flow.status.update$(cloneDest, cloneOrigin)
9:   until ($flow.allReceived(cloneDest)$ or
                                   flow.allReceived(dest))
10:  $mergeFlows(flow, clonedFlow)$
```

The BaseRoutingAlgorithm (line 3) refers to any underlying routing algorithm such as Dijkstra's shortest path algorithm or equal-cost multi-path (ECMP) algorithm, which is to be enhanced by SMART. $\mathit{SMARTRoute}$ routes the flows from the origin to the destination entirely using the BaseRoutingAlgorithm unless a threshold defined in the flow policies is met. The thresholds can be defined as system-wide policies, such as minimal throughput and latency, in network system and individual flow level. A skyline approach [2] is assumed in the presence of conflicting tenant-specific, flow-specific, or system-wide policies, to find the best possible compromise considering all the requirements. If a threshold is met (checked in line 4), the *SMART* enhancements are invoked on the flow, to mitigate the possibility of an SLA violation. Hence, the SDN controller reroutes the subflow in the new alternative route towards the clone destination, or forwards the packets of the subflow in both the original and alternative routes.

A node and a packet are chosen as the breakpoint node and packet respectively, using the markBreakPoint() invoked in line 5. Having the breakpoint node as the origin, a subflow is cloned or diverted starting from the breakpoint packet to the rest of the flow. The destination of the cloned or diverted subflow is defined as the clone destination, where the subflow is merged with the rest of the flow to reconstruct the original flow. findCloneDest() (line 6) decides the clone destination based on the flow and its status consisting of information potentially related to the policy violation. A subflow is cloned by cloneFlow() (line 7).

The status of the flow is updated to the controller through flow.status.update() (invoked in line 8) as the tags are read by the middlebox architecture. Flow status consists of the information crucial for the reconstruction of the original flow at the flow destination, such as the sequence number and the original parent flow. The cloned flow status updates enable network traffic monitoring from the controller, which can further be propagated to the presentation layer through the controller northbound API.

Flow Reconstruction: The flow reconstruction phase waits till all the packets necessary to recompose the original flow are received at the clone destination or the final destination (as checked in line 9). Once the minimum packets necessary to completely reconstruct the flow are received, the original flow is reconstructed by invoking the mergeFlows() operation (line 10). If the clone destination is different from the original destination, the recomposed flow continues in its original route towards the destination. Sequence numbers and the status indicating the parent flow from the flow packets are leveraged in reconstructing the flow. Once the original flow is received or reconstructed at the clone destination, duplicate packets are dropped on the fly.

Clone approach minimizes the extent of the necessity to reconstruct the flow; if all the packets from the original flow are received before the packets from the clone, the clone will be dropped. For the divert approach, and for the clone approach if the packets of the cloned flows arrived earlier, the flow will be reconstructed by merging the packets from the diverted or cloned subflow to the packets of the original flow that have already arrived.

The following priority flows of the same route may be replicated and rerouted, or diverted in the origin, in an alternative route. Thus, while a fraction of the initial short flows may still violate SLAs due to the time overhead imposed by the cloning and recomposing of the flows, following flows will be able to avoid the violating route altogether. When flows created by *SMART* replicate the entire flows, only the first of the flows to arrive will be considered at the destination. This replicate approach is a special case of the clone approach, where flow breakpoints and reconstructions are not applicable as there is no subflow to merge. As the replicate approach resends the entire flow from the origin to the destination in one or more alternative routes, the necessity for recomposing and packet-level manipulation is avoided, albeit with more redundancy.

SMART *Breakpoint*: Breakpoint is a pointer to the node and the flow where the subflow is cloned. The controller chooses the breakpoint dynamically, and writes rules on the breakpoint nodes to divert or clone the upcoming packets of the priority flows. Information on breakpoints are not stored statically in the flows or the controller beyond the time frame of subflow construction. Algorithm 2 elaborates on marking a breakpoint for the flow - choosing the exact breakpoint node and packet from which the flow is to be included in the diverted or cloned subflow.

Algorithm 2 Marking the Breakpoint

```
1: procedure MARKBREAKPOINT($flow$, $origin$, $dest$, $policies$, $links$)
2:   for ($link$ in $flow.route$) do
3:     if (policies.isThresholdMet()) then
4:       breakPoint.node ← current.node
5:       breakPoint.packet ← current.packet
6:       Return $breakPoint$
7:   breakPoint ← flow.estimate(policies.breakPolicy)
8:   Return $breakPoint$
```

For each flow, line 3 checks whether any of the thresholds defined in the policies is met, along the current link or the following node. If a specific node or a link is estimated to be responsible for the policy violation, the node will be marked as the breakpoint node (line 5), the current packet as the breakpoint packet (line 6), and the breakpoint will be returned (line 6). If no specific malfunctioning link or node identified, the delay is due to either i) network congestion across multiple nodes and links or ii) the flows being much larger than the typical flows in the data center and hence taking longer to complete the routing. In these cases, the breakpoints depend on policies and are decided statistically (line 7). As the origin of the diverted/cloned subflow, breakpoint node reroutes the packets to the destination in an alternative route as they arrive there. All the following packets arriving to the breakpoint node will be diverted/cloned in the alternative route(s).

***Prototype Implementation*:** A prototype of *SMART* was implemented following the devised architecture. The FlowTags controller deployed along with OpenDaylight aims to make the priority tags readable by the controller. Hence, the FlowTags controller as well as the middlebox implementations were kept minimal as a proof of concept to have a simple software middlebox to tag the flows and then to read and interpret the tags from the control plane, than a complete reimplementation of FlowTags for OpenDaylight. The data plane consisting of the nodes and middleboxes were emulated with Mininet through Python scripts. FlowTags controller, Rules Manager, and the Enhancer were deployed in the Karaf container as OpenDaylight bundles. *SMART* control plane modules were developed using Oracle Java 1.8.

III. Preliminary Assessment

A data center network with leaf-spine topology was emulated with around 1000 nodes in a distributed simulation and emulation environment, on a cluster with 6 identical nodes (Intel® Core™ i7-2600K CPU @ 3.40GHz processor and 12 GB memory). Leaf-spine topology was used as it i) offers multiple potential alternative paths between the pairs of nodes, hence satisfying the prerequisite of *SMART*, ii) often replaces the traditional tree topologies in industrial and research data center networks, and iii) is used in the related network research [3], [4]. While leaf-spine topology of common data centers have exactly two-hops, we further evaluated with extended leaf-spine topologies with longer path lengths. Figure 2 shows the time taken to route the flows with equal-cost multi-path (ECMP) as the base routing algorithm in a congested network, with and without *SMART* enhancements. *SMART* improved the performance by cloning the priority flows in an alternative route readily available in ECMP, and replicated the following priority flows of the same congested path in the new route.

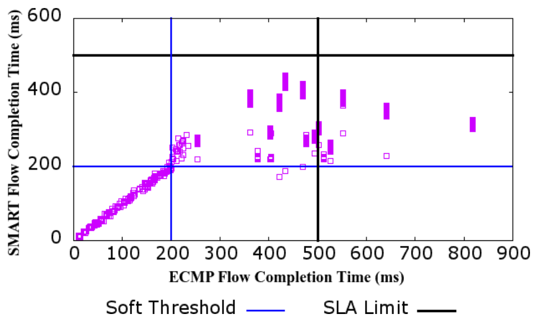

Fig. 2. *SMART* Adaptive Clone/Replicate with ECMP

SLA violations were avoided by *SMART* by up to 95%. Majority of the flows that originally violate SLAs, abide to the SLA with *SMART* enhancements. Performance of the controller and switches in detecting the violations, and updating the rules, contributes to the potential SLA violations. However, there is no flow which has an SLA violation with *SMART* enhancement, which is not also violated with the base routing. Unless the soft threshold was met, *SMART* enhancements were not invoked, as it indicated that the existing route was good enough to meet SLA and no congestion was expected.

***Assessment of Overheads*:** *SMART* exhibits an adaptive behaviour to the nature of the congestion, finding the right time to clone or divert. The contribution to congestion from cloning the subflows is minimal, as only around 16.7% of the packets of the higher priority flows were found to be cloned in the typical data center network modelled, and hence the overall redundancy will be further smaller, depending on the fraction of the flows that are considered higher priority.

The controller computations, such as determining the breakpoint node and packet, monitoring the network flows for thresholds, imposing/changing the flow tables and policies in the relevant switches, and enforcing the SLA for priority flows in the congested network based on the tags, are executed in the scale of microseconds. The overhead was estimated to be lower than 100 ms in switches when the breakpoints are manipulated and flow tables are updated, with a minimal overhead in the bandwidth. As enhancement algorithms and FlowTagger are integrated with the SDN architecture, no significant overhead was caused by the deployment of *SMART*.

Mininet emulations of an about 1000-node data center with a distributed controller deployment of OpenDaylight over 6 nodes and *SMART* enhancements showed that the controller can handle the routing, rerouting, and reconstruction of flows and subflows effectively. The majority of the decisions are handled by the nodes themselves with minimal intervention from the controller, unless a violation is triggered. Subflows still respect the ordering of packets with sequence numbers and flow IDs interpreted by *SMART* at the clone destination. Hence, reconstruction of the original flow at the destination is straightforward, dropping the duplicate packets. The enhancements are adaptive to minimize the overhead even for much smaller flows, where if the performance improvement is minimal by cloning subflows, entire following/downstream flows of the same priority, in the same path, will be replicated and routed in an alternative route along with the original route, or rerouted in an alternative route omitting the slow route.

IV. Related Work

Multipath TCP (MPTCP) is a popular extension and enhancement to TCP to use the available multiple paths between the origin and destination nodes to send a network flow [5]. MPTCP uses subflows in transferring data between the nodes in a network, where it recomposes the original flow at the destination from the subflows. While MPTCP increases the bandwidth utilization of the network and its efficiency, it is found that MPTCP can be unfair towards the TCP clients in the network [6]. Opportunistic Linked Increases Algorithm (OLIA) is proposed as an enhancement to the fairness of MPTCP, making it fair and pareto-optimal [6]. Further research on MPTCP focus on improving the MPTCP kernel implementations to match the design goal of MPTCP. MPTCP and the work that was built upon MPTCP are a major motivation behind the architecture of *SMART*, with the subflow handling mechanisms borrowed from MPTCP design.

Dynamically rerouting the network flows to optimize the bandwidth consumption has been proposed in the previous work [7]. QJump [8] is a Linux Traffic Control module that allows critical latency-sensitive applications to jump the queues in the presence of packets of lower levels, focusing a shorter flow completion time. QJump and *SMART* offer differentiated SLAs with a focus on QoS for higher priority flows through bypassing the traditional network routing, though *SMART* leverages redundancy in an adaptive manner in addition to 'jumping the queues'.

pFabric [4] finds that the increase in flow completion time of short flows is due to the waiting for long flows to complete. It focuses on optimizing the flow completion time for latency-sensitive short flows, practically ranging up to a few 10s of milliseconds. *SMART* targets the flows ranging from both very large elephant flows to small mouse flows as long as they are classified as priority flows by the tenant. Moreover, by choosing multiple alternative paths in an adaptive replicate approach, *SMART* can satisfy the short flows that have a flow completion time of a few milliseconds.

Preemptive Distributed Quick (PDQ) [9] is designed to complete flows quickly and meet the deadlines in a fair manner by following a few pre-defined procedures, enhancing the flow completion time offered by TCP. D^3 is a congestion control protocol that provides a deadline-aware alternative to TCP for data centers [10]. Conga offers congestion-aware load balancing for data center networks through flowlet switching [3]. Flowlets are defined as bursts or chunks of packets of a flow, that are separated from the other bursts of chunks by a gap [11]. Flows are often composed of flowlets and gaps between the flowlets, enabling an efficient partitioning of flows as flowlets and routing them in multiple alternative routes. Conga flowlets are similar to *SMART* subflows though Conga does not handle differentiated SLA or QoS guarantees.

V. Conclusion and Future Work

Preliminary evaluations showed the efficiency of *SMART* in offering SLA-awareness to data center networks and applications. The overhead caused by the redundancy in priority flows can be justified as typically only a small fraction of flows are of higher priority. An ongoing development effort implements *SMART* on a real data center network to evaluate against the identified related work quantitatively.

Acknowledgements: This work was supported by national funds through Fundação para a Ciência e a Tecnologia with reference UID/CEC/50021/2013 and a PhD grant offered by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC).

References

[1] S. K. Fayazbakhsh, L. Chiang, V. Sekar, M. Yu, and J. C. Mogul, "Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags," in *Proc. USENIX NSDI*, 2014.

[2] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *Data Engineering, 2001. Proceedings. 17th International Conference on*. IEEE, 2001, pp. 421–430.

[3] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, F. Matus, R. Pan, N. Yadav, G. Varghese *et al.*, "Conga: Distributed congestion-aware load balancing for datacenters," in *Proceedings of the 2014 ACM conference on SIGCOMM*. ACM, 2014, pp. 503–514.

[4] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker, "pfabric: Minimal near-optimal datacenter transport," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 435–446, 2013.

[5] S. Barré, C. Paasch, and O. Bonaventure, "Multipath tcp: from theory to practice," in *International Conference on Research in Networking*. Springer, 2011, pp. 444–457.

[6] R. Khalili, N. Gast, M. Popovic, U. Upadhyay, and J.-Y. Le Boudec, "Mptcp is not pareto-optimal: performance issues and a possible solution," in *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM, 2012, pp. 1–12.

[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *NSDI*, vol. 10, 2010, pp. 19–19.

[8] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft, "Queues don't matter when you can jump them!" in *Proc. NSDI*, 2015.

[9] C.-Y. Hong, M. Caesar, and P. Godfrey, "Finishing flows quickly with preemptive scheduling," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 127–138, 2012.

[10] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron, "Better never than late: Meeting deadlines in datacenter networks," in *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 4. ACM, 2011, pp. 50–61.

[11] S. Kandula, D. Katabi, S. Sinha, and A. Berger, "Dynamic load balancing without packet reordering," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 2, pp. 51–62, 2007.