Distributed SDN for Mission-critical Networks

Mathieu Bouet, Kévin Phemius, and Jérémie Leguay
Thales Communications & Security
4 avenue des Louvresses, 92230 Gennevilliers, France
{mathieu.bouet, kevin.phemius, jeremie.leguay}@thalesgroup.com

***Abstract*—Mission-critical networks now interconnect datacenters, enterprise, customer sites and mobile entities. They thus must be resilient, adaptable and easily extensible. The emergence of Software-Defined Networking (SDN) protocols, which enables to decouple the control plane from the data plane, opens up new ways to architect such networks. In this paper, we propose DISCO, an extensible DIstributed SDN COntrol plane able to cope with the distributed and heterogeneous nature of modern mission-critical networks. DISCO controllers manage their own network domain and communicate with each others to provide end-to-end network services. This inter-controller communication is based on a lightweight and highly manageable pub-sub mechanism used by agents to self-adaptively share aggregated local and network-wide information. We implemented DISCO on top of Floodlight, an OpenFlow controller, and the AMQP protocol. We demonstrated how DISCO's control plane dynamically adapts to heterogeneous network topologies while being resilient enough to survive to disruptions and attacks and providing classic functionalities such as end-point migration. The experimentation results we present are organized around two use cases: inter-domain connectivity disruption and migration of a virtual machine.**

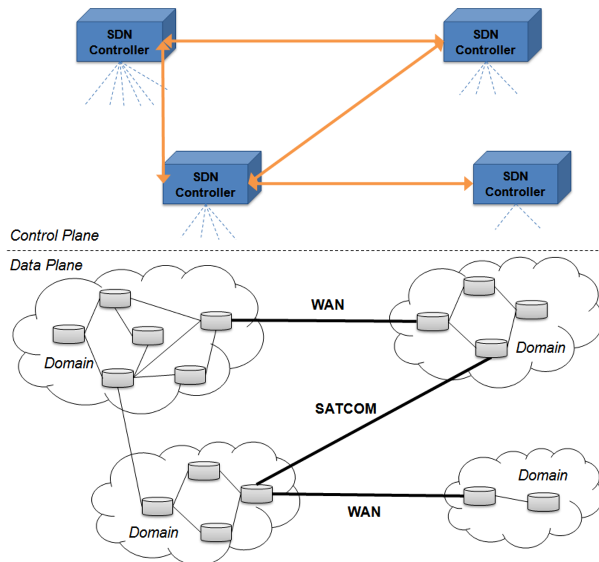

Fig. 1. Distribution of the SDN controllers over SDN domains.

I. Introduction

Resilient, scalable and extensible networks are critical to interconnect datacenters, enterprise networks and even - potentially deployable - access networks. The Software Defined Networking (SDN) paradigm has emerged from the need to overcome the primary limitations of today's networks: complexity, lack of scalability and vendor dependence. It is based on three main principles: separation of software and physical layers, logically centralized control of information and network programmability. The ability to program networks, interact with network elements and manage a unified multi-vendor multi-technology environment enables service providers and network operators to innovate faster and to reduce operational and capital expenditures. While SDN was initially used in datacenters, it is now considered for overlay networks and multi-domain networks [1].

The SDN paradigm has emerged over the past few years through several initiatives and standards, FORCES [2] being one example. The leading SDN protocol in the industry is the OpenFlow protocol. It is specified by the Open Networking Foundation (ONF) [3], which regroups the major network service providers and network manufacturers. The majority of current SDN architectures, OpenFlow-based or vendor-specific, relies on a single or master/slave controllers, that is a physically centralized entity. This centralization, adapted for datacenters, is not suitable for heterogeneous and deployable networks. In addition, the centralized SDN controller represents a Single Point Of Failure (SPOF), which makes SDN architectures highly vulnerable to disruptions and attacks [4]. The distribution of the SDN control plane has been recently addressed either with a hierarchical organization [5] or with a flat organization [6]. These approaches avoid having a SPOF and enable to scale up sharing load among several controllers. However, these distributed SDN control planes have been designed for datacenters. The controller instances share huge amount of information to synchronize the states.

In this paper, we address SDN for mission-critical networks (Fig. 1), which can be used to interconnect datacenters, enterprise networks, customer sites and mobile entities. The distributed and heterogeneous nature of these deployments call for a distributed multi-domain network control plane which should be lightweight, adaptable to user or network requirements, and robust to failures. Current state of the art distributed SDN solutions are not suitable, as they do not provide a fine grained way to control and adapt inter-controller information exchanges.

We propose DISCO, a *DIstributed SDN COntrol plane* for mission-critical networks. It relies on a per domain organization, where each DISCO controller is in charge of an SDN domain and uses a lightweight and highly manageable pub/sub mechanism to share aggregated local and network-wide information with neighbor SDN controllers. We show how DISCO dynamically adapts to heterogeneous network topologies while providing classic functionalities such as end-point migration and being resilient enough to survive to disruptions and attacks.

This work is partially supported by the DISCO project (ANR-13-INFR-013).

Contrary to state of the art distributed SDN control planes, DISCO well discriminates heterogeneous inter-domain links such as high-capacity MPLS tunnels and SATCOM interconnections with poor bandwidth and latency. We implemented DISCO on top of Floodlight [7], an OpenFlow controller, and the AMQP [8] protocol. To evaluate its performance, we show an evaluation of its functionalities on an emulated SDN according to two use cases: inter-domain topology disruption and virtual machine migration.
The rest of this paper is organized as follows. First, Sec. II presents the related work. Then, Sec. III details DISCO architecture composed of an intra-domain part and an inter-domain part Sec. IV presents the implementation. The evaluation results along the two use cases we considered are analyzed in Sec. V. Finally, Sec. VI concludes this paper.

II. Related work

Several attempts have been done to tackle the problem of scaling SDNs. A first class of solutions, such as DIFANE [9] and DevoFlow [10], address this problem by extending data plane mechanisms of switches with the objective of reducing the load towards the controller. DIFANE tries to partly offload forwarding decisions from the controller to special switches, called authority switches. Using this approach, network operators can reduce the load on the controller and the latencies of rule installation. DevoFlow, similarly, introduces new mechanisms in switches to dispatch far fewer 'important' events to the control plane.
A second class of solutions proposes to distribute the SDN controllers. HyperFlow [6], Onix [11], and Devolved controllers [12] try to distribute the control plane while maintaining a logically centralized using a distributed file system, a distributed hash table and a pre-computation of all possible combinations respectively. These approaches, despite their ability to distribute the SDN control plane, impose a strong requirement: a strongly consistent network-wide view in all the controllers. On the contrary, Kandoo [5] proposes a hierarchical distribution of the controllers based on two layers of controllers: (i) the bottom layer, a group of controllers with no interconnection, and no knowledge of the network-wide state, and (ii) the top layer, a logically centralized controller that maintains the network-wide state.
In addition, [13] analyzes the trade-off between centralized and distributed control states in SDN, while [14] proposes a method to optimally place a single controller in an SDN network.
Recently, Google has presented their experience with B4 [1], a global SDN deployment interconnecting their datacenters. In B4, each site hosts a set of master/slave controllers that are managed by a gateway. The different gateways communicate with a logically centralized Traffic Engineering (TE) service to decide on path computations. While BGP is used between border network elements to exchange routes, possibly with external service providers or operators, proprietary APIs and protocols are used between all the software pieces (gateways, controller, TE engine).
DISCO differs from state of the art solutions as it provides a distributed control plane for heterogeneous networks based on a message-oriented (pub/sub) communication bus. State of the art distributed control planes are not adaptable to heterogeneous network deployments since they impose a consistent network-wide state in all controllers and thus generate large control traffic. On the contrary, DISCO separates and aggregates local and network-wide information. Furthermore, DISCO enables to dynamically adapt to the varying networking capabilities.

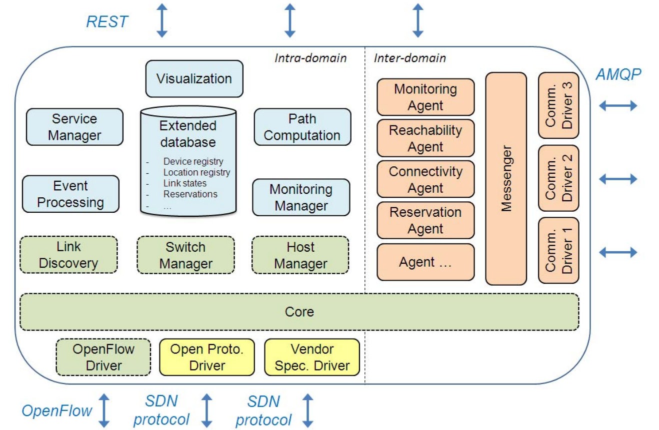

Fig. 2. DISCO Controller Architecture.

III. DISCO Architecture

A. Overall architecture

DISCO is a distributed multi-domain SDN control plane which enables the delivery of end-to-end network services. A DISCO controller is in charge of a network domain and communicates with neighbor domains to exchange aggregated network-wide information for end-to-end flow management purposes.
Fig. 2 presents the architecture. It is composed of an intra-domain part, which gathers the main functionalities of the controller, and an inter-domain part, which manages the communication with other DISCO controllers (reservation, topology state modifications, disruptions, ...). In addition to this east-west interface, a controller has at least one southbound API used to at least push policies to the network elements and retrieve their status. Finally, a northbound API enables to push management policies to the controller (e.g., service and user priorities), to manage SLAs and report network service status. A controller is composed of several modules managed by the *Core* component. It enables to start, stop, update the modules and provides them with a communication bus. Our architecture leverage from classical off-the-shelf modules that SDN controllers [7] provide such as an *OpenFlow driver* to implement the OpenFlow protocol, a *switch manager* and *host manager* to keep track of the different network elements, and *link discovery* implementing LLDP (Link Layer Discovery Protocol). We present in the rest of this section the modules that we have specifically developed for intra-domain and inter-domain flow management.

B. Intra-domain part

The intra-domain modules enable to monitor the network and manage flow prioritization so that the controller can compute the routes of priority flows based on the state of the different network parameters. The modules also enable to dynamically react to network issues (broken link, high latency,

bandwidth cap exceeded, ...) by redirecting and/or stopping traffic according to the criticality of the flow. This work extends our previous work [15] demonstrated in a centralized architecture or intra-domain context.

The *Extended Database* is a central component in which each controller stores all the intra-domain and inter-domain knowledge on network topology, monitoring and ongoing flows. All the modules and agents presented in the rest of this section either enrich or use this information, with the ultimate goals of taking actions on flows. This database and its associated model resemble to what the recent IETF group Interface to the Routing System (I2RS) [16] is trying to standardize.

The *Monitor Manager* module gathers information such as the flow throughput on the switches using the appropriate messages described in the OpenFlow protocol specification (STATISTICS_REQUEST and REPLY) [3]. In addition, this module measures the one-way latency and packet loss rate on intra-domain links by sending at a given source node specially crafted Ethernet frames including a time-stamp, and retrieving them at a given destination node to measure the elapsed time. This method has been described in [17]. By doing these operations periodically, the controller maintains an up-to-date view of link and network devices performances in the *Extended database*. For peering links with other domains, a simple Ping is used to estimate the round trip delay and packet losses.

The *Events Processing* module keeps tracks of variations or saturation events. By setting up ceiling values, the controller can immediately react if a value goes out of bound. *Events* can work with absolute values (e.g.,the total amount of dropped packets on a port) or relative values (e.g., the number of lost packets in the last second).

The *Path Computation* module computes routes for flows from source to destination using a variation of the Dijkstra algorithm, that is with QoS metrics and taking into account existing reservations. If a link on the route is considered impaired (by consulting the *Events Processor* module), a new route is computed and flow pre-emption mechanisms are applied if necessary. This module uses the *Extended Database* to retrieve information about flow descriptions (e.g., priority, bandwidth and latency constraints), network topology (e.g., capacity, routing preferences) and monitoring information.

The *Service Manager* module is responsible for the management of network SLAs inside its domain. Upon reception of a service request and all along the network lifetime, it verifies, using other modules, the feasibility and respect of SLAs. It can receive requests from the northbound API or from neighboring domains for end-to-end service provisioning.

On top of these modules, a GUI through the *Visual Manager* module allows the visualization of the network and the interaction between the user and the modules. It gather information from many other modules of the controller to display relevant information to the network operator and provides parameterization capabilities (e.g., flow priorities, new events, new routes).

C. *Inter-domain part*

A DISCO controller communicates with neighbor domain controllers to exchange aggregated network-wide information. They are composed of two key elements: (i) a *Messenger* module which discovers neighboring controllers and maintain a distributed publish/subscribe communication channel, and (ii) different *agents* that use this channel to exchange network-wide information with intra-domain modules. This way *Path Computation* can learn for instance to which neighbor domain a packet has to be routed to reach a given host.

1) Messenger: The Messenger module implements a control channel between neighboring domains. It should support group and direct communications to exchange status information (link state, host presence) and request actions (e.g., reservations) from other controllers. The usual communication patterns used by IETF protocols such as Open Shortest Path First (OSPF) protocol, Resource Reservation Protocol (RSVP) and Border Gateway Protocol (BGP) should be supported, namely step-by-step diffusion (e.g.,distance vectors), network-wide flooding (e.g., link states), uncased queries (e.g., reservation requests), and publish/subscribe messages (e.g., route updates).

To meet these requirements, we have chosen the Advanced Message Queuing Protocol (AMQP) [8] as a base for the implementation of *Messenger*. AMQP is an open standard and a thin application layer protocol for message-oriented middleware. It offers built-in features for message orientation, queuing with priority, routing (including point-to-point and publish-and-subscribe), reliable delivery and security. Due to the convergence of network and IT systems such as cloud management, AMQP is an interesting solution being lightweight, highly controllable and software-oriented. It is, for instance, used in OpenStack [18] for loosely coupled communication between the different components.

AMQP is generally used in client-server mode. This means that *Messenger* executes a server and uses clients to connect to the different servers of neighboring controllers. In this mode, *Messenger* only help local agents to exchange information with agents of neighbor domains, but does not provide communication support for network-wide exchanges. For this, it could relay the information for one domain to another by implementing its own broadcast or message forwarding mechanism. The downside of this solution is that it makes the implementation of *Messenger* more complex.

Although AMQP is generally used in client-server mode, implementations, such as RabbitMQ [19], propose a *federation* mode in which servers can be networked. In this mode, subscriptions are relayed to all the nodes in the federation and publications are routed to the right servers hosting subscribers. RabbitMQ takes care of all these operations. This means that we do not finely master the exchanges between the different AMQP brokers, which can be problematic in very constrained networks. Indeed, a message sent by a domain will be routed to every other domain interested. Eventually, the message will reach its destination but several copies will also arrive following different routes. The network footprint will thus increase, especially in a large interconnected network. Our current implementation of *Messenger* uses the federation mode for simplicity reasons, but we plan to extend it with more efficient broadcast capabilities based, for instance, on a simple spanning tree.

Messenger provides an open communication bus on top of which any agents can be plugged dynamically. It can subscribe to topics published by other agents and start publishing on any topic. Note that security mechanisms could be added to secure communication on a given set of topics or to filter publications from modules that have not been authorized.

2) Agents: To support QoS routing and reservation func-

tionalities at the inter-domain level, we have defined and implemented four main *agents*. The *Connectivity agent* is in charge of sharing with all the other domains the presence of peering links with neighboring domains. This *agent* works in an event-driven fashion as it sends information only if a new domain is discovered or a peering link changes. This information is extracted and filled up from and into the *Extended Database* of each controller, like any other information received by *agents*. This connectivity information will lately be specifically used by *Path Computation* to locally take routing decisions. The *Monitoring agent* periodically sends information on available bandwidth and latency between all the pairs of peering points to inform about the capability to support transit traffic in the domain. The *Reachability agent* advertizes on an event basis the presence of hosts in domains so that they become reachable. This service can be conceptually seen as an implementation of the Locator/Identifier Separation Protocol (LISP) [20] as it maintains at each controller a mapping between hosts and domains. The *Reservation agent* takes care, like RSVP, of inter-domain flow setup, teardown and update request including application capability requirement such as QoS, bandwidth, latency, etc. All these requests are locally handled in each domain by the *Service Manager*. *Reservation agent* uses direct communications with neighbor domains along the paths that need to be created or maintained.

Each *agent* publishes and consumes messages on a required subset of topics that they manage to ensure the consistency in the system. The exchanged information concern reachability (a list of reachable hosts in agent's domain), connectivity (a list of peering domains), and monitoring (the status of peering transit paths in terms of latency, bandwidth,...). This way, each domain controller is able to build a view of the inter-domain network and have capabilities to perform routing, path reservation and manage SLAs.

3) Interoperability and extensibility: The SDN domains managed by DISCO may communicate with other domains using classical IETF technologies. To ensure this interoperability, like in the Google deployment B4, border nodes may have to send BGP messages to exchange connectivity information. To manage this, an additional BGP agent should be added to the current architecture.

The DISCO architecture is agnostic from the SDN protocol and switches used. Despite the fact that our current implementation is OpenFlow-based, it could integrate a vender-specific southbound module.

IV. DISCO Implementation

We have implemented DISCO on top of Floodlight [7], an open source OpenFlow controller. The green hatched modules on Fig. 2 have been directly taken from Floodlight's Java source code. We have developed in Java the other software modules, except the two SDN protocol drivers in yellow that are currently empty, to manage intra-domain and inter-domain.

A. Messenger implementation

Messenger is implemented like any other Floodlight module. It subscribes to receive Packet_IN messages from the *Core* module, can write its own Packet_OUT messages, calls and stores information in the extended database and reads the Floodlight configuration file at startup. It requires the following configuration parameters: *messaging_server_type*, *messaging_server_listening_port*, and *agents_list*. The *messaging_server_listening_port* specifies the port where the *Messenger* instance can be reached. The *messaging_server_type* determines which messaging driver to use, as our architecture allows using different AMQP implementations such as RabbitMQ or ActiveMQ. Our current implementation relies on a RabbitMQ driver using AMQP in federation mode. Additional optional parameters can also be specified. *Messenger* activates *agents* from the list *agents_list*. *Agents* are small classes that handle inter-domain exchanges to and from modules managing intra-domain flows.

Messenger implements an extended version of LLDP (Link Layer Discovery Protocol), that we call Messenger-LLDP (M-LLDP), to discover neighboring domain controllers. M-LLDP messages are similar to regular LLDP messages but contain an option for OpenFlow. An Organizationally Unique Identifier (OUI) has been allocated to OpenFlow by IEEE. *Messenger* sends these messages to announce its presence on border links where other OpenFlow domains may be reached, i.e., where switches that it manages have ports leading to unknown equipment. When a reply to a discovery message is received, *Messenger* establishes an AMQP connection with its peer and stops sending discovery messages on the border link. Otherwise, *Messenger* keeps sending periodically the following M-LLDP messages with information on how to reach him (a packet of 60 Bytes, which is a relatively low network footprint):

```
0x7F (127 - LLDP's Custom TLV type)
   0x00 0x26 0xE1 (OpenFlow OUI)
     0x17 (Messenger subtype)
       0x02 (controller ID)
       0x03 (switch ID)
       0x04 (switch port)
       0x05 (server IP)
       0x06 (server port)
       0x08 (server name)
```

Messenger offers a publish/subscribe communication channel for inter-domain exchanges between *agents*. *Messenger* uses two special topics for its basic operations. First, a topic named *ID.*.** is created, ID being the identifier of the controller. This topic allows other controllers to directly send messages to it. This is used, for instance, for bandwidth reservation requests. Second, a topic named *general.*.** enables to communicate with all the other controllers in the federation. For example, it is used when a controller wants to leave a federation. This deletes the logical link between itself and the other controller and warns the *agents* to stop sending messages to this particular controller.

Messenger uses drivers to communicate with the implementation of AMQP running on the machine. Each AMQP driver must support the following set of functions:

1) subscribe (topic) and unsubscribe_topic (topic): add and delete a *topic* from the topic list that the node is interested to receive.
2) pair (neighbor controller ID) and unpair (neighbor controller ID): create and delete a inter-domain control channel with a *neighbor controller*. This function tunes the subscriptions so that a node receives or not information from this *neighbor controller*.
3) send (topic, message): send a *message* on a specific *topic* to the federation.

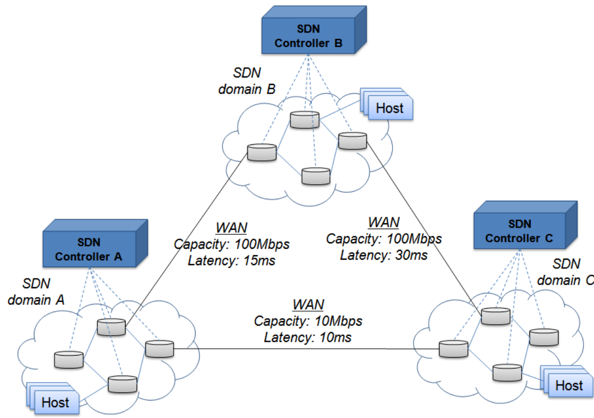

Fig. 3. A three domain SDN topology.

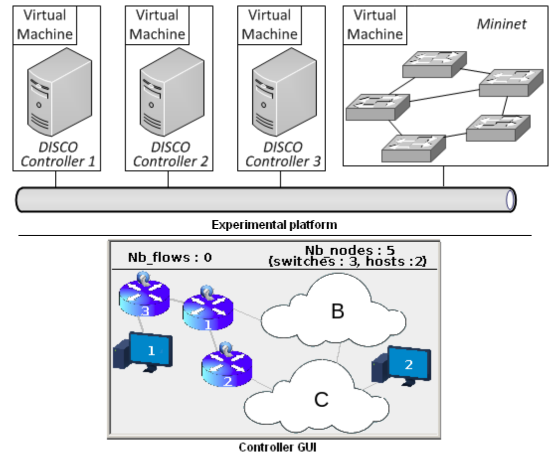

Fig. 4. GUI of Controller A (bottom) and Experimental setup (top).

Messenger also uses Keep-Alive messages every 500ms to test the presence of neighboring controllers. In case of a controller failure, the absence of 3 successful contiguous Keep-Alive responses will trigger a procedure to mitigate this failure.
The *Messenger* application has been conceived to be easily extensible. Without altering the *core* classes of Floodlight, a developer can improve it either by providing a driver for a different implementation of AMQP extending the abstract *MessengerDriver* class or add functionalities by adding another *agent*. The topic format that we use is also highly flexible as every *agent* can define its own topic. Furthermore, wildcards can be used to make subscriptions lighter and better manageable by developers.

B. Agents implementation

Agents use *Messenger* to exchange information with neighboring domains. We have implemented four agents: *Monitoring*, *Reachability*, *Connectivity* and *Reservation* (see Sec. III-C). They all publish on specific topics such as *monitoring.ID.bandwidth.2s* that the monitoring *agent* located at the controller with identifier ID advertizes every other second the remaining bandwidth that it can offer to transit traffic.
Upon reception of information from *agents* in neighboring domains, the local *agents* store them in the extended Floodlight database. This information is then used by local modules to take decisions on flows. These decisions are generally the outgoing peering link to choose for a given flow.
The *Reservation agents* implements a RSVP-like reservation protocol to provision end-to-end resources. *Agents* thus exchange reservation requests and responses with flow descriptors. Messages can be directly sent to the next domain controller on a path with the *ID.*.*.* topic.
Messenger and its dependencies (*agents*, drivers, ...) were written with just over 2400 lines of Java code. There was minimal intervention in the legacy code, except in the GUI and the Extended Database to represent remote domains. The intra-domain modules written beforehand to extend Floodlight amount to almost 12,000 lines of code. While running, *Messenger* only adds around 14MB to the total memory used by Floodlight, which is about 100MB[1].

[1]This value depends mainly on the heap size limit allocated to the JVM. It can be as low as 64MB up to several hundred Megabytes.

V. Evaluation

In addition to classic functions such as QoS routing, reservation and pre-emption, DISCO aims at being resilient to disruptions both on the control plane (e.g., controller failure, inter-controller communication failure) and on the data plane (e.g., inter-domain link failure). We have thus define two use cases that enable the evaluation of these features.

A. Testbed and setup

Fig. 3 presents the network topology considered in the performance evaluation. Each network domain A, B and C is managed by a local DISCO controller, which coordinates with its neighbor DISCO controllers. This setup is representative from a typical enterprise network where several sites (edge networks or datacenters) are interconnected with different WANs. The hosts connected to the network domains can be either user terminals or virtual machines (VM).
The testbed is enclosed in a private cloud as shown in Fig. 4 (top). The network is emulated using *Mininet* [21], a tool used to create rich topologies and instantiate Open vSwitch [22] switches and virtual hosts. *Mininet* is hosted on a dedicated VM and the controllers are hosted on separate VM. The different link latencies and bandwidths are enforced using Linux's *tc* command. This setup allows us a fine control on the network. Fig. 4 (bottom) also presents the graphical user interface of controller A showing that it has the knowledge of all the switches it manages, of all the other domains, namely B and C, and of the different hosts, local and remote.

B. Use Case 1: Adaptive information exchange

In this scenario, we show how the exchanges in the control plane can self-adapt to the network conditions. In order to reduce the network footprint of control information exchanged between domains, *agents* adopt a twofold strategy: (1) they identify alternative routes to offload traffic from weak outgoing interconnections (e.g., low-bandwidth satellite connection, congested link), and (2) they reduce the frequency of control messages for these links if they do not find an alternative route. Each *Monitoring agent* usually sends information every $2s$. This period increases to $10s$ for weak

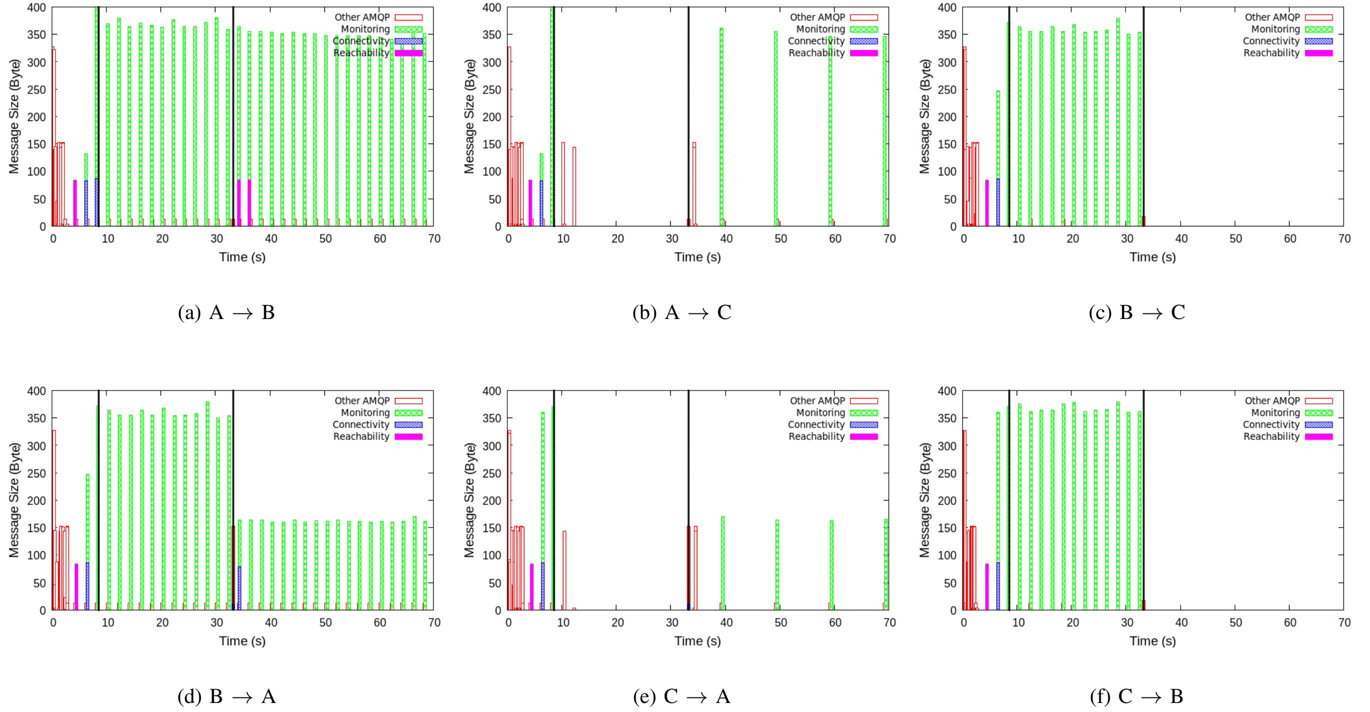

Fig. 6. Inter-controller communications. The packets come from the agents (monitoring, reachability, reservation) and AMQP itself. The bootstrap and discovery phase ends at $t = 9s$. At $t = 33s$, the link $B \leftrightarrow C$ is cut off.

interconnections. The *Connectivity* and *Reachability agents* also send their information using alternative routes whenever possible. However, contrary to the *Monitoring agents*, they only exchange messages in a reactive manner, that is when an event occurs.

Upon bootstrap and discovery, the three controllers reach the situation described on top of Fig. 5. In this scenario, the link between the domains A and C is congested. Its latency equals $\geq 50ms$. B is thus relaying control information for A and C in order to offload the congested link. In case the inter-domain link between B and C fails, *Monitoring agents* reconfigure themselves to the situation presented at the bottom of Fig. 5 where monitoring traffic is passed through the weak link $A \rightarrow C$, but at a lower frequency.

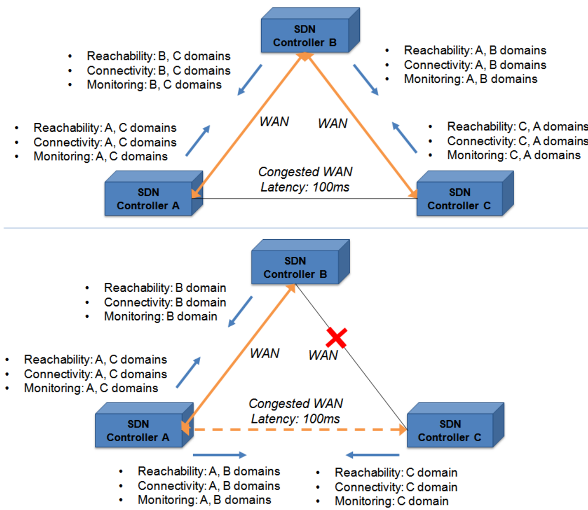

Fig. 5. DISCO controllers use high capacity inter-domain links to exchange information (top), DISCO controllers adapt the content and the frequency of their information exchanges (bottom).

Fig. 6 presents the evaluation we have conducted to show how the DISCO control plane adapts to the nominal situation and when the inter-domain link between B and C fails. This figure presents the link utilization in both directions right after the controllers discover each other and start exchanging AMQP messages. Each bar represents the TCP payload size of received packets per category[2]. This experimental scenario can be split up into three phases:

1) *Network discovery* till $t = 9s$ where controllers exchange their knowledge about hosts and the inter-domain network topology. AMQP is particularly active during this phase because the brokers have to create the federations and subscribe to the different topics. In this phase the monitoring has already started but is not yet adapted to weak links.
2) *Monitoring adaptation* from $t = 9s$ to $t = 33s$ where agents have discovered a weak link and adapt their behavior accordingly. We observe on Fig. V-B and Fig. 6(e) that monitoring is shot down after $t = 10s$ as the link $C \leftrightarrow A$ is weak (congested), while monitoring traffic increases on Fig. 6(c) and Fig. V-B.
3) *Failure recovery* starting right after we cut the link between B and C at $t = 33s$. Information is trans-

[2] In the current implementation, controllers exchange JSON messages for ease of development and integration. However, compression is planned in future releases.

mitted over the link between A and C, but with an adapted frequency as shown in Fig. 6(e) and Fig. 6(b). Monitoring traffic sent over $B \rightarrow A$ decreases as information about $B \leftrightarrow C$ is no longer necessary.

We additionally tested what would happen if a controller fails entirely. *Messenger* has a built-in feature whereupon if a controller fails 'gracefully', it can warn its neighbors so that they can prepare for the failure. Otherwise, the Keep-alive system will warn a controller if its neighbor is no longer reachable. In that case, the logical control plane links are severed, no messages are carried any more toward this controller and other failure mitigation processes occur (e.g., if the fallen domain was used to exchange messages between two domain, they will reconnect by other path if available).

C. Use Case 2: Migration of a virtual machine

In this scenario, a virtual machine is migrated from one domain to another by a cloud management system. Upon the detection of a new MAC or IP address in a domain, the *Reachability agent* sends an update message to announce that it now manages this new host. Other controllers update the mapping that they maintain locally between hosts and domains so that their path computation module can route flows towards this new location. To speed up the convergence of the handover, they also immediately update all the rules related to this host in the switches that they manage if a flow going toward this IP was already established. To test this case, we launched a 10 Mbits/s UDP flow from a host located in domain A to another situated in C during 40s. According to the values presented in Fig. 3, this flow is taking all of the available bandwidth on the $A \leftrightarrow C$ link whose lantecy is equal to 10ms. If the cloud management system moves the Virtual Machine from domain C to B, C will announce that it is no longer capable of reaching the VM while B reports that it is now handling it following the migration. These messages, along with the reconfiguration of the mapping between host and domain, will allow A to dynamically reroute the flow to its new destination.

Fig. 7 shows the performance in terms of service interruption and network metrics. The VM is now reachable through the $A \leftrightarrow B$ link which has a lower latency. We can see the latency drops at the moment of the change in the flow's path. We ran this experiment ten times to get average values. The switch is not instantaneous; it takes on average $117ms$ with some packet losses (1.26% in average). In any case, this experiment shows the adaptability of DISCO in a environment where end-hosts can move between domain and their communication can be seamlessly rerouted by the controllers.

VI. Conclusions and Perspectives

We have proposed DISCO, a *DIstributed SDN COntrol plane* for mission-critical networks. It relies on a per domain organization where each controller is in charge of an SDN domain and share aggregated local and network-wide information with the neighbor controllers. We have implemented DISCO on top of the Floodlight [7] and the AMQP protocol [8]. We have evaluated its functionalities according to two use cases: inter-domain topology disruption and migration of a virtual machine. The results show how DISCO dynamically adapts to heterogeneous network topologies and is resilient to attacks and disruptions. Contrary to state of the art distributed SDN control planes, DISCO well discriminates heterogeneous inter-domain links such as high-capacity MPLS tunnels and SATCOM interconnections. Future work includes a failover mechanism in order for SDN controllers to automatically take the control of the switches when a neighbor controller fails.

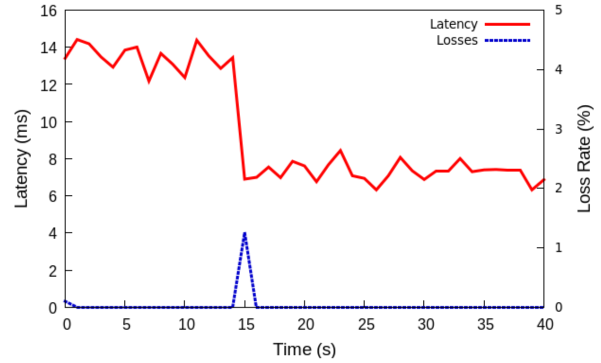

Fig. 7. Impact on flow latency and loss rate when moving $host2$ from $domainC$ to $domainB$.

References

[1] S. Jain and al., "B4: Experience with a Globally-Deployed Software Defined WAN," in *ACM SIGCOMM*, 2013.

[2] L. Y. et al., "Forwarding and Control Element Separation (ForCES) Framework," RFC 3746, Apr. 2004.

[3] "Open Networking Foundation (ONF)." [Online]. Available: http://www.opennetworking.org/

[4] D. Kreutz, F. Ramos, and P. Verissimo, "Towards secure and dependable software-defined networks," in *ACM HotSDN*, 2013.

[5] S. H. Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *ACM HotSDN*, 2012.

[6] A. Tootoonchian and Y. Ganjali, "Hyperflow: a distributed control plane for openflow," in *INM/WREN*, 2010.

[7] "Floodlight OpenFlow Controller." [Online]. Available: http://floodlight.openflowhub.org/

[8] "AMQP." [Online]. Available: http://www.amqp.org

[9] M. Yu *et al.*, "Scalable flow-based networking with difane," in *SIGCOMM Comput. Commun. Rev. 40, 4*, 2010.

[10] A. Curtis *et al.*, "Scalable flow-based networking with difane," in *ACM SIGCOMM*, 2011.

[11] T. Koponen *et al.*, "Onix: a distributed control platform for large-scale production networks," in *OSDI*, 2010.

[12] A.-W. Tam, K. Xi, and H. Chao, "Use of devolved controllers in data center networks," in *IEEE INFOCOM workshops*, 2011.

[13] D. Levin, A. Wundsam, B. Heller, N. Handigol, and A. Feldmann, "Logically centralized?: state distribution trade-offs in software defined networks," in *ACM HotSDN*, 2012.

[14] B. Heller, R. Sherwood, and N. McKeown, "The controller placement problem," in *SIGCOMM Comput. Commun. Rev. 42*, 2012.

[15] K. Phemius and M. Bouet, "Implementing OpenFlow-based resilient network services," in *IEEE CLOUDNET*, 2012.

[16] "Interface to the Routing System (I2RS) WG," IETF, 2013.

[17] K. Phemius and M. Bouet, "OpenFlow: Why latency does matter," in *IFIP/IEEE IM*, 2013.

[18] "OpenStack." [Online]. Available: http://www.openstack.org

[19] "RabbitMQ." [Online]. Available: http://www.rabbitmq.com

[20] F. et al., "Locator/ID separation protocol (LISP)," RFC 6830, Apr. 2010.

[21] "Mininet." [Online]. Available: http://mininet.org

[22] "Open vSwitch." [Online]. Available: http://openvswitch.org/