DISCO: Distributed Multi-domain SDN Controllers

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

***Abstract*—Software-Defined Networking (SDN) is now envisioned for Wide Area Networks (WAN) and constrained overlay networks. Such networks require a resilient, scalable and easily extensible SDN control plane. In this paper, we propose DISCO, an extensible *DIstributed SDN COntrol plane* able to cope with the distributed and heterogeneous nature of modern overlay networks. A DISCO controller manages its own network domain and communicates with other controllers to provide end-to-end network services. This east-west communication is based on a lightweight and highly manageable control channel. We implemented DISCO on top of the Floodlight OpenFlow controller and the AMQP protocol and we evaluated it through an inter-domain topology disruption use case.**

I. Introduction

The SDN paradigm has emerged over the past few years through several initiatives and standards from the need to overcome the primary limitations of today's networks. It is based on three main principles: separation of software and physical layers, centralized control of information and network programmability. SDN is now envisioned for multi-datacenter environments [1] and WANs.
The leading SDN protocol in the industry is the OpenFlow protocol. The majority of current SDN architectures, OpenFlow-based or vendor-specific, relies on a single or master/slave controllers, that is a physical centralization. Adapted for datacenters, they are not suitable for multi-technology and wide networks. In addition, the centralized SDN controller represents a Single Point Of Failure (SPOF), which makes SDN architectures highly vulnerable to disruptions and attacks [2].
Recently, proposals have been made to distribute the SDN control plane (Sec. II). These approaches avoid having a SPOF and enable to scale up sharing load among several controllers. However, mainly designed for datacenters, they generate huge amount of data to synchronize controllers (eg. distributed database).
In this paper, we propose DISCO, a *DIstributed SDN COntrol plane* for WAN and overlay networks. It relies on a per domain organization, where each controller is in charge of an SDN domain, and provides a lightweight and highly manageable inter-controller channel. This channel enables *agents* to share aggregated network-wide information and hence support end-to-end network services. We demonstrate how DISCO dynamically adapts to heterogeneous network topologies while being resilient enough to survive to disruptions and attacks. 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 the Floodlight [3] OpenFlow controller and the AMQP [4] protocol and evaluated its functionalities on an emulated SDN.
The rest of this paper is organized as follows. First, Sec. II analyzes related work. Then, Sec. III presents DISCO architecture. Our implementation is explained in Sec. IV and the evaluation in Sec. V. Finally, Sec. VI concludes this paper.

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

II. Related work

Several attempts have been done to distribute SDN controllers. HyperFlow [5] and Onix [6] propose to distribute the control plane while maintaining a logical centralization using a distributed file system and a distributed hash table respectively. These approaches, despite their ability to distribute the SDN control plane, impose a strong requirement: a consistent network-wide view in all the controllers. On the contrary, Kandoo [7] 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. Recently, Google has presented their experience with B4 [1], a global SDN deployment interconnecting their datacenters with a centralized Traffic Engineering service and clusters of controllers in each data center.
In addition, [8] analyzes the trade-off between centralized and distributed control states in SDN, while [9] proposes a method to optimally place a single controller in an SDN network.
DISCO differs from state of the art solutions as it provides a distributed control plane for WAN and constrained networks based on a message-oriented communication bus. State of the art distributed control planes are not adaptable to heterogeneous and constrained (bandwidth, latency, ...) network deployments. Most of them impose a consistent network-wide state in all controllers and thus generate large control traffic.

III. DISCO Architecture

A DISCO controller (Fig. 1) is composed of two parts: 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, monitoring, ...). In addition to this east-west interface, a controller has at least one southbound SDN interface used to push policies to the network elements and retrieve their status and a northbound interface to receive management policies (e.g., service and user priorities) and report network service status.

A. Intra-domain functionalities

The intra-domain modules enable to monitor the network and manage flow prioritization so that the controller can

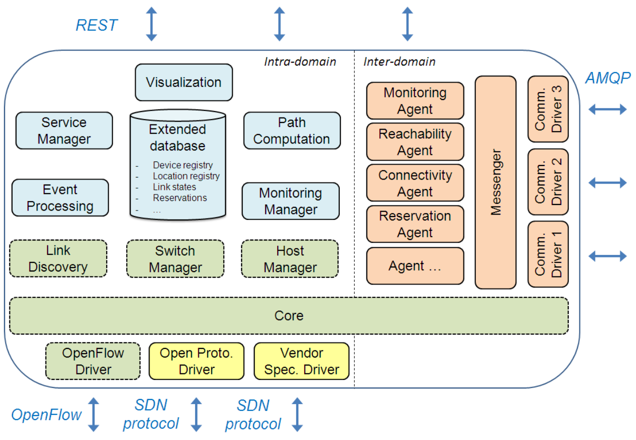


Fig. 1. DISCO Controller Architecture.

compute the paths 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 flows. This work extends our previous work [10] on a centralized architecture, that is intra-domain context. The central component is the *Extended Database* 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 either enrich or use this information.

B. *Inter-domain functionalities*

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 other controllers.

1) Messenger: This module provides 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 should be supported: 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) [4] 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.
Each DISCO controller has both an AMQP server in order to publish information and an AMQP client to retrieve information from other controllers. This way, each controller manages its topics and its subscriptions. *Messenger* thus provides an open communication bus on top of which any agents can be plugged dynamically.

2) Agents: To support network-wide functionalities, we have defined and implemented four main *agents*. The *Connectivity agent* is in charge of sharing peering link details with all the other domains. This *agent* works in an event-driven mode as it sends information only if a new domain is discovered or a peering link changed. This information is extracted and filled up from and into the *Extended Database* of each controller, like any other information received by *agents*. 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. 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.
Each *agent* publishes and consumes messages on a required subset of topics that they manage through *Messenger* to ensure the consistency in the system. The exchanged information concerns reachability (a list of reachable hosts in agent's domain), connectivity (a list of peering domains), and monitoring (the status, latency and bandwidth of peering links). 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.

IV. DISCO Implementation

We have implemented DISCO on top of Floodlight [3], an open source OpenFlow controller. The green hatched modules on Fig. 1 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 application. It subscribes to receive Packet_IN messages from the *Core* module, writes its own Packet_OUT messages, calls and stores information in the extended database. It currently relies on a RabbitMQ driver using AMQP in federation mode. *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 peer controller.
Messenger uses drivers to communicate with different implementations of AMQP. 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

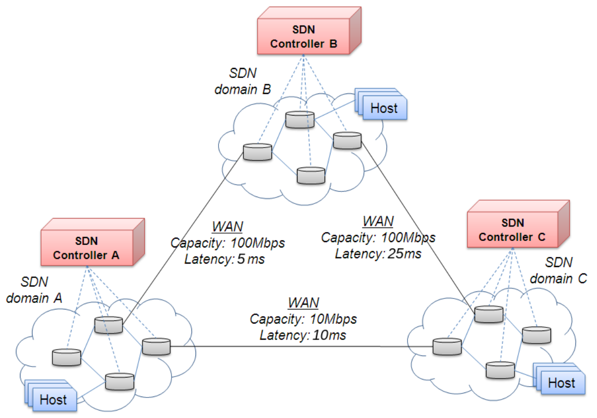


Fig. 2. Multi-domain SDN Topology.

receive.
2) pair (neighbor controller ID) and unpair (neighbor controller ID): create and delete a inter-domain control channel with a *neighbor controller*.
3) send (topic, message): send a *message* on a specific *topic*. *Messenger* also uses Keep-Alive messages every 500ms to test the presence of neighboring controllers.

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-B). They all publish on specific topics. For example, the monitoring *agent* of the controller whose identifier is *ID* advertizes every *2* seconds on the topic *monitoring.ID.bandwidth.2s* the remaining bandwidth that it can offer to transit traffic . The *Reservation agent* implements a RSVP-like reservation protocol to provision end-to-end resources. Such *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. The intra-domain modules written beforehand to extend Floodlight amount to almost 12,000 lines of code.

V. Evaluation

A. Testbed and setup

Fig. 2 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. 3. The network is emulated using *Mininet* [11], a tool used to create rich topologies and instantiate Open vSwitch switches and virtual hosts. The different link latencies and bandwidths are enforced using Linux's *tc* command. This setup allows us a fine control on the network.

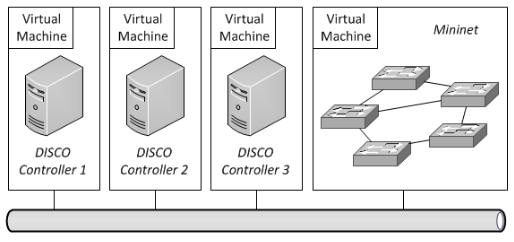


Fig. 3. Experimental setup.

B. Evaluation: 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 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. 4. 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. 4 where monitoring traffic is passed through the weak link $A \rightarrow C$, but at a lower frequency.
Fig. 5 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

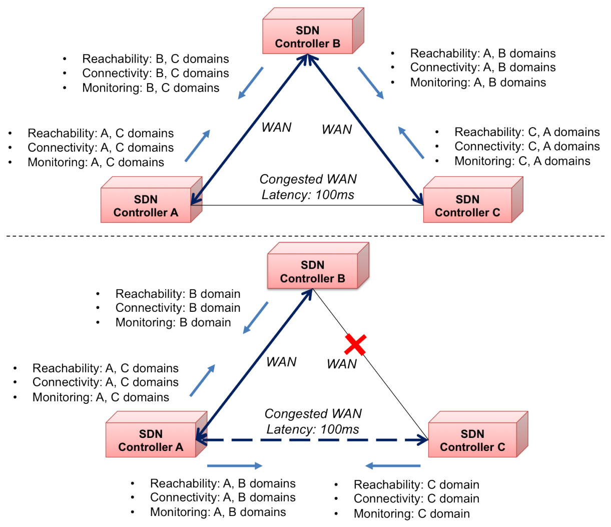
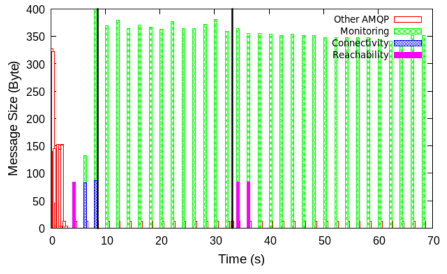
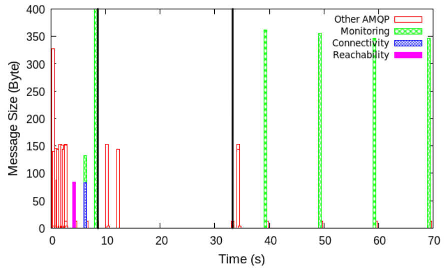


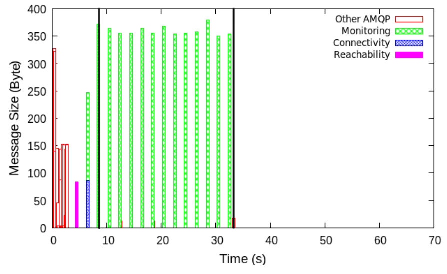
Fig. 4. Adaptable information exchange: (top) Congested situation: DISCO controllers use high capacity inter-domain links to exchange information; (bottom) Inter-domain link disruption: DISCO controllers adapt the content and the frequency of their information exchanges.



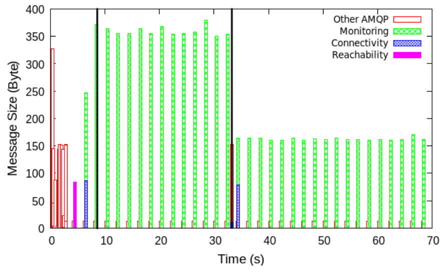
(a) A $\rightarrow$ B



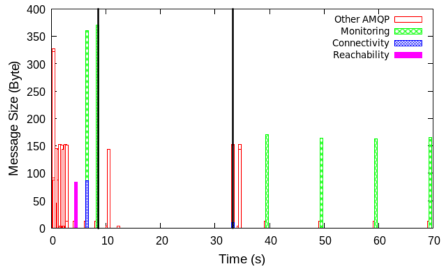
(b) A $\rightarrow$ C



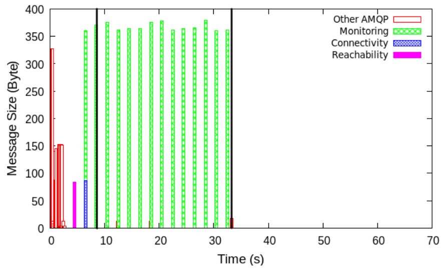
(c) B $\rightarrow$ C



(d) B $\rightarrow$ A



(e) C $\rightarrow$ A



(f) C $\rightarrow$ B

Fig. 5. Adaptive information exchange on the different links. Packets come from the different agents and AMQP itself. At $t = 9s$, the bootstrap and discovery phases end. At $t = 33s$, the link $B \leftrightarrow C$ is cut off.

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[1]. 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. V-B that monitoring is shot down after $t = 10s$ as the link $C \leftrightarrow A$ is weak (congested), while monitoring traffic increases on Fig. V-B and Fig. V-B.
3) *Failure recovery* starting right after we cut the link between B and C at $t = 33s$. Information is transmitted over the link between A and C, but with an adapted frequency as shown in Fig. V-B and Fig. V-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).

[1]Controllers currently exchange JSON messages for ease of development and integration. Compression is planned in future releases.

VI. Conclusion

We have proposed DISCO, a *DIstributed SDN COntrol plane* for WAN and constrained overlay networks. It relies on a per domain organization, where each controller is in charge of an SDN domain, and provides a lightweight and highly manageable inter-controller channel. We demonstrated how DISCO dynamically adapts to heterogeneous network topologies while being resilient enough to survive to disruptions. More details can be found in [12]. Our future works include the extension of DISCO with additional resilient and recovery mechanisms so that a controller can take the control of switches from a neighbor domain on the fly in case of failure.

References

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

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

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

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

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

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

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

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

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

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

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

[12] K. Phemius, M. Bouet, and J. Leguay, "DISCO: Distributed Multi-domain SDN Controllers," *CoRR*, vol. arxiv.org/abs/1308.6138, 2013.