ENDEAVOUR: Towards a flexible software-defined network ecosystem

<table>
<thead>
<tr>
<th>Project name</th>
<th>ENDEAVOUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project ID</td>
<td>H2020-ICT-2014-1 Project No. 644960</td>
</tr>
<tr>
<td>Working Package Number</td>
<td>2</td>
</tr>
<tr>
<td>Deliverable Number</td>
<td>2.2</td>
</tr>
<tr>
<td>Document title</td>
<td>Design and final requirements of the SDN architecture</td>
</tr>
<tr>
<td>Document version</td>
<td>0.8</td>
</tr>
<tr>
<td>Editor in Chief</td>
<td>Chiesa, UCLO</td>
</tr>
<tr>
<td>Authors</td>
<td>Bruyere, Canini, Chiesa, Nguyen</td>
</tr>
<tr>
<td>Date</td>
<td>28/01/2016</td>
</tr>
<tr>
<td>Reviewer</td>
<td>Castro, QMUL</td>
</tr>
<tr>
<td>Date of Review</td>
<td>19/01/2016</td>
</tr>
<tr>
<td>Status</td>
<td>Public</td>
</tr>
</tbody>
</table>
## Revision History

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Description</th>
<th>Author</th>
</tr>
</thead>
<tbody>
<tr>
<td>23/12/15</td>
<td>0.1</td>
<td>First draft</td>
<td>Canini, Chiesa, Nguyen</td>
</tr>
<tr>
<td>24/12/15</td>
<td>0.2</td>
<td>Umbrella contribution</td>
<td>Bruyere, Nguyen</td>
</tr>
<tr>
<td>28/12/15</td>
<td>0.3</td>
<td>iSDX contribution</td>
<td>Canini, Chiesa</td>
</tr>
<tr>
<td>04/01/16</td>
<td>0.4</td>
<td>ez-Segway contribution</td>
<td>Canini, Nguyen</td>
</tr>
<tr>
<td>10/01/16</td>
<td>0.5</td>
<td>Final pre-revision changes</td>
<td>Chiesa, Nguyen</td>
</tr>
<tr>
<td>14/01/16</td>
<td>0.6</td>
<td>Executive summary and introduction</td>
<td>Canini, Chiesa, Nguyen</td>
</tr>
<tr>
<td>19/01/16</td>
<td>0.7</td>
<td>Review</td>
<td>Castro</td>
</tr>
<tr>
<td>24/01/16</td>
<td>0.8</td>
<td>Revision of contents based on internal review</td>
<td>Canini, Chiesa, Nguyen</td>
</tr>
</tbody>
</table>
Executive Summary

Routing in the Internet is limited by the standard de-facto inter-domain routing protocol (i.e., BGP). Routes are only based on the IP destination prefix and are exchanged in a distributed way among neighboring networks. Software-Defined Networking (SDN) has reshaped the design of many networks and is poised to enable new capabilities in inter-domain traffic delivery. A natural place for this evolution to occur is at Internet Exchange Points (IXPs), which are becoming increasingly prevalent. Because many Autonomous Systems (ASes) interconnect at IXPs, introducing flexible control at these locations makes it easier for them to control how traffic is exchanged in a direct, and more fine-grained way.

In this document, we present an overview of the SDN-enabled ENDEAVOUR IXP architecture, the first SDX architecture that can operate at the scale of the largest IXPs. The ENDEAVOUR architecture offers to the network operators a single, coherent, global interface to interact with the underline network that can be leveraged by network operators to develop novel applications on top of its architecture, hence boosting innovation in the inter-domain Internet ecosystem. The ENDEAVOUR architecture is designed to support arbitrarily fine-grained routing policies that can be defined by the IXP participants via an intuitive, high-level policy specification language. Our preliminary simulations against a trace from one of the largest IXPs in the world found that ENDEAVOUR can compile a realistic set of policies for 500 IXP participants in less than three seconds.
Contents

1 Introduction 8

2 Overview of the ENDEAVOUR Architecture 8
   2.1 System Design .................................................. 9
      2.1.1 IXP Fabric ................................................. 10
      2.1.2 Fabric Manager ............................................. 10
      2.1.3 Participant Information Base ......................... 12
      2.1.4 Network State Monitoring .............................. 12
      2.1.5 Applications ............................................. 12
      2.1.6 Route Server/Relay ....................................... 13
   2.2 Scalability Challenges ....................................... 14
   2.3 Reliability Challenges ....................................... 16

3 Scalable Routing 18
   3.1 Edge Switches .................................................. 18
      3.1.1 Background ................................................. 19
      3.1.2 Partitioning Control-Plane Computation ............... 22
      3.1.3 Distributing Forwarding Rules & Tags ................. 24
      3.1.4 Decoupling SDN Policies from Routing ............... 26
   3.2 Preliminary Simulations at the Edge Switches .......... 29
      3.2.1 Experiment Setup ......................................... 29
      3.2.2 Steady-State Performance ................................ 31
      3.2.3 Runtime Performance ..................................... 33
   3.3 Core Switches .................................................. 36
      3.3.1 No More Broadcast Traffic ............................... 36
      3.3.2 Towards a Segment Routing-Like Approach .......... 39
      3.3.3 Umbrella Interaction with Route Servers .......... 41
   3.4 Preliminary Simulations at the Core Switches .......... 42

4 Reliable Routing 48
   4.1 The Network Update Problem ................................. 49
   4.2 The Idea ........................................................ 50
      4.2.1 Decentralizing for Fast Consistent Updates .......... 51
      4.2.2 Dealing with Deadlocks ................................ 53
   4.3 Ez-Segway ...................................................... 55
      4.3.1 Flow Segmentation ........................................ 55
      4.3.2 Scheduling Updates ..................................... 58
      4.3.3 End-Point Policy Coherence ....................... 61
List of Figures

1. ENDEAVOUR architecture for an SDN-enabled IXP ........................................... 9
2. A simplified IXP topology. ................................................................................... 21
3. Reachability and Next Hops (in bold) for AS A and AS B. .................................. 21
4. The matrix representation. .................................................................................. 22
5. Partitioning the Control-Plane Computation. ...................................................... 23
6. Distributing forwarding rules and tags. ................................................................. 23
7. Reduction of flow rules with reachability encoding. ............................................. 27
8. Number of forwarding table entries. ..................................................................... 30
9. Number of virtual next-hop IP addresses for centralized and distributed control planes. ................................................................. 32
10. Time to perform policy compression. ................................................................. 32
11. Rate at which forwarding table entries are updated. ......................................... 33
12. Compute time for increasing forwarding actions. ................................................. 34
13. Compute time for increasing sustained rates of BGP updates. ......................... 34
14. Rate at which a participant’s border router receives gratuitous ARPs. ............. 35
15. Handshake in case of ARP request. ..................................................................... 37
16. Typical topology of a medium to large IXP. ........................................................ 38
17. Example of multi-hop in the core. ...................................................................... 39
18. Example of Umbrella forwarding scheme. ......................................................... 40
20. Average number of rules required at the edge. ..................................................... 44
21. Average number of hops in large European exchange points. ......................... 45
22. Effect of an ARP-induced broadcast storm on a peering router CPU occupancy. ......................................................................................................................... 47
23. BGP state machine evolution during the ARP-induced broadcast storm. .......... 48
24. CPU performance among legacy fabrics, legacy fabric with ARP-sponge, and Umbrella ................................................................................................................. 49
25. An example network update. .............................................................................. 51
26. A network update with splittable deadlock. ......................................................... 54
27. A network update with segment deadlock. .......................................................... 54
28. Different cases of segmentation. ......................................................................... 56
29. Choosing a correct update operation. ................................................................. 60
30. Dependency graph of deadlock cases. ................................................................. 60
31. Dependency graph of deadlock cases. ................................................................. 61
32. Update time of ez-Segway versus a Centralized approach for real WAN topologies: B4 and Internet2. ................................................................................. 63
Update time (without computation) of ez-Segway versus a centralized approach for various settings. .......................... 64

List of Tables

1 Three distributed SDX Controllers. .............................. 29
2 Core Switch Flow Table example. ............................... 37
1 Introduction

As set forth in Deliverable 2.1, the ENDEAVOUR IXP aims to support a large variety of network applications (e.g., fine-grained inbound and outbound policy specification, Traffic-Engineering, or DDoS mitigation) built on top of a centralized fabric manager layer, which offers a single, coherent, global view of the network.

At the core of the ENDEAVOUR architecture, Software-Defined Networking (SDN) plays a crucial role. SDN is deemed by the networking community a revolutionary paradigm for controlling, orchestrating resources and routing within a network. There are two main reasons: (1) it defines a clear programmable interface towards the data-plane level and (2) it allows, in its most deployed implementation (i.e., OpenFlow [27]), to route traffic at fine-grained scale. It is therefore natural and extremely appealing for an SDN-enabled IXP to leverage the SDN programmability abstraction in order to bring the fine-grained capabilities to its participants in the form of a configurable inbound/outbound policy specification language. As such, ENDEAVOUR allows IXP participants to overcome the inherent limits of traditional BGP while boosting routing innovation in the monolithic inter-domain ecosystem.

In this context, the design of the ENDEAVOUR architecture aims at supporting the following identified requirements: (i) enable the definition of arbitrarily fine-grained routing policy specification by the IXP participants, (ii) realize flexible routing policies with high efficiency, reliability and scalability, especially at the data-plane level, and (iii) provide basic primitive functions that seamlessly update the forwarding state of the IXP network without creating forwarding anomalies (e.g., forwarding loops).

In Section 2, an overview of the ENDEAVOUR architecture is presented. Section 3 and Section 4 describe the main scalability and reliability challenges that the ENDEAVOUR architecture solves, and presents our preliminary simulation results.

2 Overview of the ENDEAVOUR Architecture

In this document, we provide an overview of the ENDEAVOUR architecture for an IXP based on the SDN paradigm.

In Section 2.1, we describe the main functionalities provided by the ENDEAVOUR IXP architecture and the interaction among the components of the whole system. In Section 2.2 and Section 2.3, we overview the
main scalability and reliability challenges that arise from implementing these functionalities, respectively. A detailed explanation of our ongoing implementation is deferred to Section 3 and Section 4.

2.1 System Design

According to the SDN paradigm, in the ENDEAVOUR architecture the control-plane functionalities are moved from the physical switches to a logically centralized controller, which is responsible of computing the forwarding state. The SDN approach introduces a clear separation between data-plane (i.e., forwarding packets) and control-plane (i.e., installing the forwarding state) functionalities. As such, by decoupling the control-plane functionalities from the physical switches to a logically centralized controller, the network architecture achieves high network modularity, which, in turn, leads to higher flexibility and eases innovation. Moreover, the SDN data plane brings programmability that enables fine-grained routing decisions, filtering and monitoring so that IXP members can override default BGP
behavior. The SDN control plane lowers management complexity and enables new services, which can be realized through programs running at the controller or interfaces exposed to the IXP members.

In the following subsections, we describe the main components of the high-level architecture, which are depicted in Figure 1.

2.1.1 IXP Fabric

At the physical level, the forwarding functions within the IXP fabric are performed by a set of SDN switches that are interconnected to each other according to the most suitable network topology (e.g., full-mesh, or spine-leaf), which typically depends on the specific characteristics of an IXP (e.g., number of inter-connected sites and geographical constraints among them). The SDN switches are responsible for forwarding packets according to their own forwarding state (i.e., the set of forwarding rules installed in each switch). The computation of the routing functions is part of the control-plane functionalities. The SDN switches can be accessed via the OpenFlow driver, which works as an adapter for communicating with the SDN switches based on the OpenFlow protocol specification [27]. Communication messages include for instance flow entry modifications, per-port statistics queries, and group tables modification.

2.1.2 Fabric Manager

At the core of the ENDEAVOUR architecture, the fabric manager is designed to ease management and to support a broad set of operations that would be extremely difficult, if not impossible, to implement in today’s IXP fabric. The fabric manager lowers management complexity and enables new services, which can be realized through programs running at the controller or interfaces exposed to the IXP participants.

By acting as an interface to the physical network, the fabric manager provides the IXP operators with a high-level interface that can be used to deploy customized applications on top of it. Namely, the fabric manager exposes the network operator to a logical view of the physical network topology, presents a coherent and global picture of the network state, and allows the operator to interact with the switches via a set of high-level primitives.

The fabric manager abstracts the details of the underlying switch hardware and OpenFlow messages from the participant and the IXP controllers and also ensures isolation between participants. The fabric
manager is based on Ryu SDN controller [41].

Such primitives can, for instance, allow the network operator to move from one forwarding state to another one without creating routing anomalies (e.g. forwarding loops, blackholes) or allow IXP participants to define their own inbound/outbound policies via a virtual-switch interface. Roughly speaking, IXP operators leverage the network controller the same way programmers interact with the operating system.

The fabric manager consists of three components: (i) the edge forwarding handler, called iSDX, which is responsible for determining the egress port of each incoming packet at the IXP based on BGP routes and custom participants policies, (ii) the core forwarding handler, called Umbrella, which is responsible for computing the paths within the IXP network, and (iii) the network update handler, called ez-Segway, which is responsible for seamlessly modifying the forwarding state of the network.

**iSDX.** The industrial Software Defined eXchange (iSDX)\(^1\) takes as input the high-level routing policies of each participant and the BGP route announcements, and outputs the forwarding state (at the edge of the network) needed to route packets according to the policies and BGP best path selection. Namely, for any packet that enters the IXP network, iSDX computes the egress port at the IXP where the packet should be forwarded to. An overview of iSDX can be found in Section 2.2, while a detailed description can be found in Section 3.1.

**Umbrella.** Forwarding within the IXP is based on source-routing techniques, in which the forwarding path within the IXP network is encoded in the header of a packet by means of a sequence of port identifiers that has to be traversed by the packet itself. We leverage the destination MAC address to encode this information, although other techniques are also viable (e.g., MPLS). An overview of Umbrella can be found in Section 2.2, while a detailed description can be found in Section 3.3.

**ez-Segway.** Whenever a routing policy changes or the IXP needs to update the forwarding state of its fabric, the routing functions have to be updated. The forwarding-state update component performs this operation by coordinating the transition phase among all the involved physical switches while avoiding routing anomalies (e.g., forwarding loops) and minimizing

\(^1\)A full description of the iSDX design and implementation will appear at the 13th USENIX Symposium on Networked Systems Design and Implementation (NSDI '16) [7].
congestion. An overview of ez-Segway can be found in Section 2.3, while a
detailed description can be found in Section 4.

As described, the components are complementary in that they absolve
to different functions. Moreover, the architecture does not mandate that
all components are used in all scenarios. For instance, for a smaller IXP
the iSDX component could be unused; then the fabric manager relies on the
Core component for forwarding packets across the IXP.

2.1.3 Participant Information Base

The IXP infrastructure is mostly “static”: when a new IXP participant
device is connected to the IXP fabric, first, its MAC address is
communicated to the IXP operator, then, the IXP operator assigns a
physical port to that specific device. This mutual coordination, which
is required between the IXP and its participants, breaks the premises of
traditional neighbor discovery mechanisms, such as ARP or ND, where
new devices can be connected to a network without any coordination
with the network administrator. For efficiency, the ENDEAVOUR IXP
controller exploits this “static” nature of the IXP environment by accessing a
Participant Information Base (PIB) that stores where IXP members devices
are located and what their MAC addresses is. By doing so, a static routing,
as described in the fabric-manager component description, can be achieved.

2.1.4 Network State Monitoring

This component is responsible of collecting routing statistics (e.g., packet
drops, traffic statistics) and offer a coherent global view of the network state.
This part of the architecture is covered by the WP3 and it is not described
in this document, unless it is relevant for understanding some parts.

2.1.5 Applications

Both the IXP operators and the IXP participants can write their own
applications that interface with the IXP controller. In our vision, the
SDN-enabled IXP fabric is a concrete opportunity for routing innovation
in the inter-domain routing. Novel applications can easily be deployed
by writing software that interacts with the coherent view offered by the
controller (rather than the obscure and indirect mechanism
offered by traditional IXP fabric). Some examples of applications of
ENDEAVOUR interests that meet the requirements set from the operational
feedback described in WP4 are Traffic-Engineering (outbound/inbound), DDoS mitigation, high-level policy specification, a Graphical User Interface (GUI) for configuring routing policies at the IXP. However, we do not mandate specific applications nor we mandate that participant applications run on IXP infrastructure. Participant applications may run on participant controllers that accesses the fabric manager via available interfaces.

**SDX policy interface.** Participants of the IXP can enforce their routing policies in two ways: by means of the standard BGP mechanism (i.e., BGP communities, AS-Path prepending, MED and so) or by leveraging a policy interface based on previous work [8]. While the first option guarantees backward compatibility with the traditional IXP design, the second option allows IXP participants to define arbitrary fine-grained routing policies in an intuitive and high-level policy specification language based on a “Per-participant Virtual Switch” abstraction. Namely, each IXP participant is given the illusion of having a virtual SDN switch at the edge of its network that is connected to all the virtual switches of every other participant. As such, IXP participants can configure it in order to perform inbound/outbound traffic engineering, DDoS mitigation, access-control-list (ACL) definition, and more. This abstraction eases the process of ensuring isolation between different participants when high level policies are transformed into forwarding rules. It is crucial to observe that the routing policies defined using this interface are visible to the IXP. As such, IXP participants whose routing policies are considered private information should not reveal this information when interacting with the SDX policy interface.

### 2.1.6 Route Server/Relay

The Route Server, which is the traditional BGP computational unit that is deployed in current IXP, to ease the exchange of BGP routes among the participants, can be either integrated within the Fabric Manager, in which case it acts as a Route Relay that forwards all the route information to the Fabric Manager, or it can be run outside the IXP fabric as a standard Route Server.

Our IXP controller is backward compatible with traditional BGP. That is, the IXP fabric can still handle interactions with IXP members that are running traditional protocols (i.e., BGP), although specific SDN advanced features will not be supported. This can be done by integrating a Route Server within the network controller.
2.2 Scalability Challenges

In the following, we show how the two forwarding components iSDX and Umbrella of the ENDEAVOUR Fabric Manager tackle the main scalability challenges that arise at the edge and the core of an SDN-enabled IXP, respectively.

**Scalability issues.** Forwarding packets with zero or minimal disruption is the ultimate goal of any network infrastructure. Given the size of the global Internet routing, which counts more than 500K IPv4 prefixes, naively supporting arbitrarily inbound/outbound policies using one single forwarding table has one main disadvantage: since it requires to compute the cartesian product of all the inbound and outbound policies, the size of the forwarding table would easily explode. Even worse, since in this case inbound policies are tightly coupled with outbound policies, i.e., in the forwarding table each inbound policy is replicated for each outbound policy, a change of a single inbound policy of an IXP participant causes an update of a large number of rules. In addition, since forwarding rules depend on whether an IXP announced or not an IP prefix, every time there is a BGP event (e.g. a set of IP prefixes announced/withdrawn), the IXP may need to dynamically recompute the forwarding state for that set of specific IP prefixes. Since the number of rules in the forwarding table is huge, the computational time of this operation becomes prohibitive for any reasonable real-world operational scenario.

**The ENDEAVOUR solution.** In the ENDEAVOUR architecture, we aim to both provide a fine-grained forwarding platform among the IXP participants, which allows network operators to gain more control in the path selection process, and to define both inbound and outbound policies. To scale in the size of the forwarding state, without limiting arbitrary fine-grained routing, we leverage source-routing techniques, which is part of the Umbrella component, in which the forwarding path within the IXP network is encoded in the header of a packet by means of a sequence of port identifiers that has to be traversed by the packet itself. Leveraging source-routing in an IXP brings several advantages:

- Complex routing operations are moved to the edge of the network, while the core retains its simplicity.
- Limited amount of information is stored in the core forwarding tables.
• Possibility of service chaining, that is, forwarding a packet through a sequence of middle-boxes.

• Easy way to debug the data-plane as the path is encoded in the packet.

We use the MAC address field of a packet to encode the path within an IXP network. We refer the reader to Section 3.3 for a more detailed explanation about the encoding of the path.

While source-routing is responsible for moving a packet from the ingress port of the IXP network to the egress port, the ENDEAVOUR data-plane still needs to compute the best egress port based on BGP routing information and the inbound/outbound policies of the IXP participants. Namely, the egress ports of a packet depends on the sender of a packet, the header of the packet (i.e., a TCP source/destination ports, IP source/destination addresses), the available valid BGP routes, the outbound policy of the sender of the packet, and the inbound policies of the IXP participants that are announcing to the sender an IP prefix for that packet. All this information must be stored in the forwarding tables in order to compute the correct egress port when a packet enters the IXP network.

To curb the explosion of rules in the data-plane state, the ENDEAVOUR architecture leverages two sophisticated techniques, which are part of the iSDX component [7].

First, different forwarding tables are used to store the inbound and the outbound policies of all the participants. This technique, which is typically known as “normalization” of data tables in the database community, avoids the problem of the forwarding state explosion and removes the coupling between inbound and outbound policies. Multiple forwarding tables are supported by the OpenFlow standard (since version 1.1).

Second, to remove any dependency of the forwarding state from the BGP reachability information that is available at a certain time, the ENDEAVOUR forwarding scheme encodes information about the IP prefix reachability through the IXP participants in the MAC address of each packet. This hence has the benefit that it drastically reduces the number of updates at the data-plane level. Namely, the MAC address contains (in a compressed way) the set of IXP participants that announced an IP prefix for the IP destination of the packet. We refer to these encoded MAC addresses as Virtual MAC (VMAC) addresses (see Section 3.1 for a more detailed explanation). In order to learn these VMACs, the IXP controller uses BGP announcements. For each IP prefix, the IXP associates a virtual IP next-hop address, whose ARP resolution translates to the VMAC containing the IXP participants that announced that IP prefix. As such, IXP participants learn
the VMACs associated with an ARP request for the virtual IP next-hops received in the BGP announcements from the IXP controller.

To support backward compatibility with legacy MAC addresses, the ENDEAVOUR architecture uses a special bit in the MAC address for distinguishing legacy from virtual MAC addresses. Source-routing MAC addresses are distinct from other MAC addresses since they are used for forwarding only in the internal network. Namely, if a packet enters the IXP network, the MAC address is either a legacy or a virtual one. If a packet arrives at an IXP switch from another IXP switch, then it is a source-routing encoded MAC address. Per-incoming-port matching is supported by the OpenFlow standard.

To summarize our forwarding scheme, we use a two-phase forwarding approach, which separates external and internal routing at an IXP. When a packet from an IXP participant arrives at the edge switch of the IXP since it enters the network from a port connected to a switch outside the IXP, it contains either a legacy or a virtual MAC address. In the first case, its MAC address identifies a specific egress port of the IXP network. As such, its MAC address is replaced by the encoded source-routing MAC address that describes the path to reach the egress point. In the second case, the MAC address is a virtual one and the packet is processed by the outbound forwarding table, which determines the best outgoing IXP participant, and consequently by the inbound forwarding table, which selects the best egress port based on the inbound policies of the receiver of the packet. Once the egress port is chosen, the MAC address is replaced by the encoded source-routing MAC address that describes the path to reach the egress point. Finally, before the packet leaves the IXP network, its correct MAC address is rewritten in its header so that it can be correctly received by the network device of the destination participant.

Observe that the ENDEAVOUR architecture adopts a “smart” edge and a simple core infrastructure, wherein the edge switches are responsible for matching packets to their outbound/inbound forwarding policies while the core switches are only responsible for forwarding them to their egress ports in the IXP.

2.3 Reliability Challenges

Reliability issues. Packet forwarding is a fundamental activity of networks to transfer the packet from one switch to another. This activity

---

2By design, the second least significant bit of the first byte of the MAC address is always zero for legacy MAC address. We set this bit to 1 for virtual MACs.
is reliable when all packets are forwarded without any forwarding failure such as routing anomalies (e.g., forwarding loops, black-hole), congestion or policy violations. A forwarding failure may occur due to either a violation of routing policies or the failure of communication links or switches. While the latter requires restoring the forwarding state in reaction to a failure detector, the former must be avoided proactively to prevent disruptions. We focus on the design of an architecture that avoids forwarding failures of routing policy enforcement. Specifically, as already set forth in Deliverable D2., the ENDEAVOUR architecture guarantees four reliability properties defined as follows:

- **black-hole freedom**: for any flow, no packet is unintentionally dropped in the network;
- **loop-freedom**: no packet should loop in the network;
- **congestion-freedom**: no link has to carry a traffic greater than its capacity;
- **policy coherence**: during network state updates, every packet should be forwarded according to either the old or new policy, but not a mixture of the two policies as this can lead to security violations or other forwarding inconsistencies.

**The ENDEAVOUR solution.** The violation of forwarding policies could be the result of (1) the invalidity of the routing policy itself or (2) the inconsistency during the transition of two forwarding policies [14], an usual activity or event in network management. While the first scenario can be easily handled by a validation to check the routing policy before applying it to the network, the second scenario is more complicated because routing policy evolves constantly during the transition. ENDEAVOUR addresses this second scenario by proposing a decentralized network update scheduling scheme. In contrast with prior methods [14, 19], which utilizes the controller to propose a centralized approach, we investigate the prospect of delegating the responsibility of consistent updates to the data plane in order to reduce the workload at the controller and speed up the update. Based on a relaxed requirement of per-packet consistency while still enforcing the same endpoint policy (henceforth called segmentation), we propose a distributed network update architecture wherein the controller is only responsible for computing the intended network configuration. The actual update function is realized by the switches, which schedule and coordinate the execution of an update.
for the entire network using partial knowledge and direct message passing. This allows every switch to update its local forwarding rules as soon as the update dependencies are met (i.e., when a rule can only be installed after dependent rules are installed at other switches). We instantiate our design in ez-Segway, a reliable update scheduling mechanism that can run as software on switches. Our algorithm enables decentralized network updates that preserve the endpoint policy while avoiding any forwarding failures and scheduling deadlocks.

We show how this approach leads to faster network updates, reduces the complexity of the scheduling computation, and constitutes the basis for a new class of algorithms that perform planned updates while reacting to accurate data plane measurements and conditions (e.g., precise link congestion [10], connectivity failures [20,21] or actual FIB update delays [10,17,39]). By delegating more responsibility to the switches, our algorithm proposes an effective mechanism to update the network.

# 3 Scalable Routing

In this section, we show how the ENDEAVOUR architecture attains fine-grained routing policy specification capabilities while minimizing both the computation time of the forwarding tables and their size.

In Section 3.1, we show how the iSDX component can efficiently implementing inbound/outbound matching forwarding rules that determine the egress IXP port of each packet that enters the IXP network. In Section 3.1, we show how packets are sent within the IXP network, throughout its core, via the Umbrella component.

## 3.1 Edge Switches

In this section, we highlight the main challenges that must be tackled in order to support arbitrary inbound/outbound policies. These operations are performed at the edge of the network when a packet enters the IXP network. In the ENDEAVOUR architecture, the edge forwarding operations are performed by a component that we call the industrial Software Defined eXchange (iSDX) [7]. As discussed in the overview section (Section 2.2), we address both control-plane and data-plane issues, that is, reducing the time for computing the forwarding state and limiting the amount of forwarding state rules installed and updated in the switches, respectively.

In Section 3.1.1, we provide a background of the SDX platform presented in [8], which we build upon. One of the techniques employed by SDX to
attain scalability is the concept of Forwarding Equivalence Class (FEC). An FEC is a set of IP prefixes that share the same forwarding behavior throughout the SDX fabric. Intuitively, by identifying the minimal number of FECs, SDX seeks to reduce the size of forwarding state. In Section 3.1.2, we describe how to partition the control-plane computation among the IXP participants so that the computation time of the Forwarding Equivalent Classes (FECs) is reduced alongside the size of the forwarding tables. In Section 3.1.3, we show how to reduce the dependency of inbound and outbound routing policies in order to reduce the time needed for updating the network after a routing policy change. In Section 3.1.4, we show how to decouple BGP reachability information from the SDN forwarding, which removes the need to update the forwarding state every time there is a BGP routing change event. In Section 3.2, we show our preliminary simulation of the ENDEAVOUR architecture with respect to the aforementioned scalability challenges.

3.1.1 Background

Brief overview of Software-Defined-eXchanges (SDXs). An SDX is an IXP consisting of a programmable SDN fabric, coupled with a BGP route server (which allows IXP participants to exchange reachability information via BGP) and an SDN controller (which allows participants to override default BGP routing behavior with more fine-grained SDN policies). The SDX controller provides each participant AS with the abstraction of a dedicated switch that it can program using match-action policies to control traffic flows. Participants may express SDN policies on both their inbound and outbound traffic; the SDX controller ensures that no SDN policy results in traffic being forwarded to a neighboring AS that did not advertise a BGP route for the prefix that matches the packet’s destination IP address.

Each participant runs an SDN control application on the central controller and has its border router exchange BGP update messages with the IXP’s route server. The SDN controller combines the SDN policies from all participants, reconciles the resulting policy with the BGP routing information, and computes and installs the resulting forwarding table entries in the IXP fabric. To avoid having forwarding entries for all prefixes, the original SDX design relied on the participants’ border routers to tag packets entering the IXP fabric with a forwarding equivalence class of destination prefixes with the same forwarding action. For backwards compatibility, the tag was the destination MAC address, set in response to the border router sending an ARP query for the next-hop IP address from.
the BGP route advertisement. The SDX route server computed a different (virtual) next-hop IP address for each equivalence class of prefixes to trigger the border router to use a common MAC address for packets sent to the group of destination IP addresses.

**Example operation.** Figure 2 shows an example topology with five participants; Figure 3 shows the routes advertised to A and B and the BGP routes that they select for each prefix (in bold). Both A and B express outbound policies. To ensure that SDN policies cause the IXP to forward traffic in a way that is consistent with the advertised BGP routes, the SDX controller *augments* each outbound policy with the reachability information. Intuitively, augmentation restricts forwarding policies so that traffic is forwarded only on paths that correspond to BGP routes that the participant has learned.

For example, suppose that A has the following outbound policies:

\[
\begin{align*}
\text{dPort}=443 & \rightarrow \text{fwd}(C) \\
\text{dPort}=22 & \rightarrow \text{fwd}(C) \\
\text{dPort}=80 \land \text{sIp}=10/24 & \rightarrow \text{fwd}(D) \\
\text{dPort}=80 \land \text{sIp}=40/24 & \rightarrow \text{fwd}(D)
\end{align*}
\]

These policies forward traffic based on values of packet header fields, overriding BGP behavior. For instance, the first policy specifies HTTPS traffic (dPort=443) should be forwarded to C. Without augmentation, A would also forward the HTTPS traffic destined for prefix P5 to C, even though C never advertised a path for P5 to A. In our example, A’s policies are then augmented as follows:

\[
\begin{align*}
\text{dIp} \in \{P1, P2, P3, P4\} \land \text{dPort}=443 & \rightarrow \text{fwd}(C) \\
\text{dIp} \in \{P1, P2, P3, P4\} \land \text{dPort}=22 & \rightarrow \text{fwd}(C) \\
\text{dIp} \in \{P1, P2, P3, P4, P5\} \land \text{dPort}=80 \land \text{sIp}=10/24 & \rightarrow \text{fwd}(D) \\
\text{dIp} \in \{P1, P2, P3, P4, P5\} \land \text{dPort}=80 \land \text{sIp}=40/24 & \rightarrow \text{fwd}(D)
\end{align*}
\]

Augmentation enforces that the destination IP (dIp) matches one of the prefixes that either C or D announces to A, therefore ensuring congruence with BGP routing. Observe that a straightforward realization of this policy requires one distinct match-action rule for each of the four prefixes. Hence, the augmented policies would result in 18 forwarding rules instead of the four rules necessary to implement the original policy.

Similarly, if B’s outbound policy is:
Figure 2: Example Topology. An example with five IXP participants. Two participants AS A and AS B have outbound policies. The other three advertise five IP prefixes to both these participants.

Figure 3: Reachability and Next Hops (in bold) for AS A and AS B.

\[
dPort=443 \rightarrow \text{fwd}(E)
\]

the SDX controller augments the policy, doubling the number of necessary rules, as follows:

\[
dIp \in \{P4, P5\} \land dPort=443 \rightarrow \text{fwd}(E)
\]

To better illustrate the scalability challenge, we capture the expansion of the switch forwarding tables using an augmentation matrix (Figure 4, left matrix). In this matrix, a row labeled as SDN_{X,Y} refers to an SDN policy written by X that results in traffic being forwarded to Y, while columns refer to IP prefixes. The value of an element \((i, j)\) indicates the number of forwarding table entries (i.e., match-action rules) in participant \(i\)'s policy.
where prefix $j$ appears. Similarly, $BGP_{X,Y}$ indicates whether $X$ selects $Y$ as the next hop for some BGP-advertised prefix, and element $(i,j)$ is 1 if participant $A$ selects the route advertised by $B$ for the prefix corresponding to column $j$.

For example, the element in row $SDN_{A,C}$ and column $P1$ reflects the fact there are two forwarding table entries that correspond to prefix $P1$: one for traffic with $dPort=443$ and one for traffic with $dPort=22$. The same applies for columns $P2$, $P3$, and $P4$. We can determine the total number of forwarding table entries (and the number contributed by each participant) by summing up the corresponding elements in the matrix. We will use this notation to describe compression techniques (and their effects) throughout the document.

### 3.1.2 Partitioning Control-Plane Computation

To achieve greater compression of the rule matrix, we need to reduce the constraints that determine how prefixes belong to the same Forwarding Equivalence Class (FEC). Rather than computing one set of equivalence classes for the entire SDX, we compute separate FECs for each participant. We first discuss how partitioning by participant reduces the size of the rule matrices and, as a side benefit, allows for faster computation. We then describe how we use multiple match-action tables and ARP relays to further improve scalability, setting the stage for further optimizations.
Partitioning the FEC Computation. Figure 5 shows similar compression and compilation steps as the ones done in Figure 4, with the important distinction that it takes place on behalf of participant A only; similar operations take place on behalf of other participants. Figure 5 highlights two important benefits of partitioning the computation of FEC across participants:

- Computing separately for each participant reduces the number of next-hops, leading to a smaller number of larger forwarding equivalence classes. In Figure 5, the number of columns reduces from five to two.

- The computational complexity of computing FECs is proportional to the number of rows times the number of columns in the rule matrix. Now, each rule matrix is smaller, and the computation for different participants can be performed in parallel.

In practice, the SDX controller could compute the FECs for each participant,
or each participant could run its own controller for computing its own FECs (as shown in Figure 1).

### 3.1.3 Distributing Forwarding Rules & Tags

In addition to computing the FECs for each participant, the system must realize these policies in the data plane.

**Decomposing the IXP fabric into four tables.** To forward traffic correctly, an SDX must combine the inbound and outbound policies for all of the participants. Representing the combination of policies in a single forwarding table, as in an OpenFlow 1.0 switch, would be extremely expensive. Some existing SDN controllers perform this type of composition [29, 43]—essentially computing a cross product of the constituent policies—and, in fact, our original SDX followed this approach [8]. Computing the cross product leads to an explosion in the number of rules, and significant recomputation whenever one of the participant policies changes.

Fortunately, modern switches have multiple stages of match-action tables, and modern IXPs consist of multiple switches. The system design capitalizes on this trend. The main challenge is to determine how to most effectively map policies to the underlying tables.

A strawman solution would be to use a two-table pipeline, where packets first enter an outbound table implementing outbound policies for the participant where the traffic originates, followed by an inbound table that applies inbound policies for the participant that receives the traffic as it leaves the IXP fabric. Using only two tables, however, would mean that some of these tables would need to be much larger; for example, the outbound table would need to represent the cross product of all input ports and outbound policies. Additionally, using only two tables makes it more difficult to scale-out the system as the number of participants grows.

As such, our design incorporates an input table, which handles all the incoming traffic and tags it with a new source MAC address based on the packet’s incoming port, so that packets can be multiplexed to the outbound table. As the packet leaves the system, it passes through an output table, which looks up the packet’s tag in the destination MAC field and both performs the appropriate action and rewrites the packet’s destination MAC address. Separate input and output tables provide a cleaner separation of functions between the modules that write to each table, avoid cross-product explosion of policies, and facilitates scale-out by allowing the inbound
and outbound tables to reside on multiple physical switches in the IXP infrastructure.

Figure 6 shows how the IXP fabric forwards a packet, while distributing the compilation and compression of policies across separate tables. Based on the destination IP address of the packet, suppose that AS A’s controller selects a route to the packet’s destination via AS D; this route will correspond to a next-hop IP address. AS A’s controller will make a BGP announcement advertising this path. AS A’s router will issue an ARP query for the advertised next-hop IP address, and its controller will respond via the ARP relay setting a virtual MAC address (in Figure 6 “VMAC-1”) as the packet’s destination MAC address.

When the packet enters the IXP fabric, the input table matches on the packet’s incoming port and rewrites the source MAC address to indicate that the packet arrived from AS A (“SRC_A”). If A has an outbound policy, the packet will match on (“SRC_A”), and the outbound table will apply an outbound policy. If A has no outbound policy for this packet, the input table forwards the packet directly to the inbound table without changing the destination MAC. This bypass is not strictly necessary but avoids an additional lookup for packets that do not have a corresponding outbound policy. A’s outbound policy thus overwrites default BGP forwarding decision and modifies the destination MAC address to “C”. The inbound table rewrites the tag to correspond with the final disposition of the packet (“C1” or “C2”), which is implemented in the output table. The output table also rewrites the tag to the receiver’s physical MAC address before forwarding.

Reducing ARP traffic overhead. Partitioning the FEC computation reduces the number of FECs per participant, but may increase the total number of FECs across all participants (i.e., the number of columns across all rule matrices). To reduce the size of the forwarding tables, each data packet carries a tag (i.e., a virtual MAC address) that identifies its FEC. The participant’s border router learns the virtual MAC address through an ARP query on the BGP next-hop IP address of the associated routes. The use of broadcast for ARP traffic, combined with the larger number of next-hop IP addresses, could overwhelm the border routers and the IXP fabric. In fact, today’s IXPs are already vulnerable to high ARP overheads [5].

Fortunately, we can easily reduce the overhead of ARP queries and responses, because each participant needs to learn about only the virtual MAC addresses for its own FECs. As such, the SDX can turn ARP traffic into unicast traffic by installing the appropriate rules for handling ARP
traffic in switches. In particular, each participant’s controller broadcasts a gratuitous ARP response for every virtual next-hop IP address it uses; rules in the IXP’s fabric recognize the gratuitous ARP broadcasts and ensure that they are forwarded only to the relevant participant’s routers. Participants’ routers can still issue ARP queries to map IP addresses to virtual MAC addresses, but the fabric intercepts these queries and redirects them to an ARP relay to avoid overwhelming other routers.

3.1.4 Decoupling SDN Policies from Routing

To ensure correctness, any SDX platform must combine SDN policies with dynamic BGP state: which participants have routes to each prefix (i.e., valid next-hop ASes for a packet with a given destination prefix), as well as the next-hop AS to use for each prefix (i.e., the outcome of BGP decision process). The large number of prefixes and participants creates scalability challenges with respect to forwarding table sizes and update rates, before SDN policies even enter the equation.

**Idea: statically encode routing.** To reduce the number of rules and updates, we develop a new encoding scheme that is analogous to source routing: The IXP fabric matches on a tag that is provisioned by a participant’s SDX controller. To implement this approach, we optimize the tag that the fabric uses to forward traffic (as described in the previous “Partitioning the FEC Computation” paragraph in Section 3.1.2) to carry information about both the next-hop AS for the packet (as determined by the best BGP route) and the ASes who have advertised routes to the packet’s destination prefix. If no SDN policy matches a packet, the system can simply match on the next-hop AS bits of the tag to make a default forwarding decision. As before, the sender discovers this tag via ARP.

To implement default forwarding, the IXP fabric maintains static entries for each next-hop AS which forward to participants based upon the next-hop AS bits of the tag. When the best BGP routes change, the entries need not change, rather the next-hop AS bits of the tags change.

To account for changes in available routes, policy entries which reroute to some participant X confirm X has advertised a route before rerouting. The method of checking for X in the tags is nearly static, meaning that in contrast to our previous design [8], BGP updates induce effectively zero updates in the IXP switch data plane. Instead, BGP updates result in tag changes, and the participant’s border router learns these dynamic tags via ARP.
We now describe how the system embeds both the next-hop AS (i.e., from the best BGP route) and the reachability information (i.e., the set of ASes that advertise routes to some prefix) into this tag.

**Next-hop encoding.** The next-hop information denotes the default next-hop AS for a packet, as determined by BGP. In the example from Section 3.1.1, A’s next-hop AS for traffic to P1 as determined by the best BGP route is D. The system allocates bits from the tag (i.e., the virtual MAC, which is written into the destination MAC of the packet’s header) to denote this next-hop. If no SDN policy overrides this default, the system applies a default priority prefix-based match on these bits to direct traffic to the corresponding next-hop. This approach reduces the forwarding table entries in a participant’s outbound table, since additional entries for default BGP forwarding no longer need to be represented as distinct entries in the forwarding table. Encoding the next hop information in this way requires log₂(N) bits, where N is the number of IXP participants. At a large IXP with up to 1024 participants, ten bits can encode information about default next-hop ASes, leaving 37 bits.

**Reachability encoding.** We now explain how to encode reachability information into the remaining 37 bits of the destination MAC address. We first present a strawman approach that illustrates the intuition before describing the scalable encoding.

**Strawman encoding.** Suppose that for a given tag, the i-th bit is 1 if that participant learns a BGP route to the corresponding prefix (or prefixes) via next-hop AS i. Such an encoding would allow the IXP fabric to efficiently

---

3The OpenFlow 1.3 standard supports this feature, which is already implemented in many hardware switches (e.g., [31]).

4One of the 48 bits in the MAC header is reserved for multicast.
determine whether some participant could forward traffic to some next-hop AS $i$, for any $i$ at the IXP. Considering the example in Section 3.1.1, $A$'s outbound policies are:

- $\text{dMac = XX...X} \land \text{dPort}=443 \rightarrow \text{fwd}(C)$
- $\text{dMac = XX...X} \land \text{dPort}=22 \rightarrow \text{fwd}(C)$
- $\text{dMac = XXX...X} \land \text{dPort}=80 \land \text{sIP}=10/24 \rightarrow \text{fwd}(D)$
- $\text{dMac = XXX...X} \land \text{dPort}=80 \land \text{sIP}=40/24 \rightarrow \text{fwd}(D)$

where $X$ stands for a wildcard match (0 or 1). This encoding ensures correct interoperation with BGP, yet we use just four forwarding table entries, which is fewer than the 18 required using augmentation (from the original example in Section 3.1.1).

Hierarchical encoding. The approach consumes one bit per IXP participant, allowing at most for only 37 IXP participants. To encode more participant ASes in these 37 bits, we divide this bitspace hierarchically. Suppose that an IXP participant has SDN policies that refer to $N$ other IXP participants (i.e., possible next-hop ASes). Then, all of these $N$ participants need to be efficiently encoded in the 37-bit space, $B$. We aim to create $W$ bitmasks \{${B_1, B_2, \ldots, B_W}$\} that minimize the total number of forwarding table entries, subject to the limitations of the total length of the bitmask.

Given $M$ prefixes and $N$ IXP participants, we begin with $M$ bitmasks, where each bitmask encodes some set of participants that advertise routes to some prefix $p_i$. We greedily merge pairs of sets that have at least one common participant, and we always merge two sets if one is a subset of the other. Iterating over all feasible merges has worst-case complexity $O(M^2)$; and there may be as many as $M - 1$ merge actions in the worst case. Each merge has complexity $O(N)$, which gives us an overall worst-case running
Table 1: Three distributed SDX Controllers.

<table>
<thead>
<tr>
<th></th>
<th>MDS</th>
<th>NH Encoding</th>
<th>Reachability Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>iSDX-D</td>
<td>✔</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>iSDX-N</td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td>iSDX-R</td>
<td>✗</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

Given 37 spare bits in the destination MAC for reachability encoding, if a participant has defined SDN policies for more than 37 participants who advertise the same prefix, then the number of bits required to encode the reachability information will exceed 37. Our analysis using a dataset from one of the largest IXPs in the world found that the maximum number of participants advertising the same prefix was only 27, implying that largest bitmask that this encoding scheme would require is 27 bits. There were 62 total bitmasks, meaning 6 bits are required to encode the ID of a bitmask, requiring a total of 33 bits for the encoding. Using a different (or custom) field in a packet header might also be possible if these numbers grow in the future.

3.2 Preliminary Simulations at the Edge Switches

We now demonstrate that iSDX can scale to the forwarding table size, data plane update rate, and control plane computation requirements of an industrial-scale IXP. Table 1 summarizes the three different iSDX designs that we compare to previous approaches: iSDX-D applies the same MDS compression technique as in our previous work [8], but with tables distributed across participants; iSDX-N additionally encodes the next-hop AS in the tag; and iSDX-R encodes both the next-hop AS and BGP reachability information in the tag.

Figure 8 summarizes our results: iSDX reduces the number of forwarding table entries for an industrial-scale IXP by three orders of magnitude as compared to an unoptimized, centralized SDX design; and by more than two orders of magnitude over the state-of-the-art SDX design [8]. This section explains these results in detail.

3.2.1 Experiment Setup

We use data sets from one of the largest IXPs worldwide, which interconnects more than 600 participants, who peer with one another via a BGP route server [37]. We had access to the RIB table dump collected from the IXP’s
route server on August 8, 2015 for 511 IXP participants. These datasets contain a total of 96.6 million peering (i.e., non-transit) routes for over 300,000 distinct prefixes. We also use a trace of 25,676 BGP update messages from these participants to the route server for the two hours following the collection of this RIB table dump (the participants’ RIBs are naturally not perfectly aligned, since dumping a BGP table of about 36 GB from the router takes about fifteen minutes). Our data set does not contain any user data or any personal information that identifies individual users. We run our experiments on a server installed at this IXP configured with 16 physical cores with 3.4 GHz and 128 GB of RAM.

This IXP does not use a programmable IXP fabric, so we assume how participants might specify SDN policies, as described in Section 3.1.1. Specifically, each participant has between one to four outbound policies for each of 10% of the total participants. The number of policies and set of participants are chosen uniformly at random. Our sensitivity analysis on this percentage shows that our results are influenced in magnitude but the underlying trends remain. Note that this setup is more taxing than the one in our previous work where only 20% of the total participants had any SDN policies at all. We also evaluate iSDX’s performance for smaller IXPs by selecting random subsets of IXP participants (ranging from 100 to 500).
500 ASes) and considering only the RIB information and BGP updates for those participants. We also repeated experiments using public RIB dumps and BGP updates collected by RIPE’s RIS servers from 12 other IXPs. As the observed workload was much smaller in this case, we omit these results for brevity.

3.2.2 Steady-State Performance

We first evaluate the steady-state performance of iSDX. To do so, we use the RIB dumps to initialize the SDX controller (multiple of them for the distributed case) and evaluate the overall performance in terms of the efficiency of data-plane compression, and the time to compile policies and compress them into smaller forwarding tables.

**Efficiency of compression.** Figure 8 shows the number of forwarding table entries for the three distributed controllers: iSDX-D, iSDX-N, and iSDX-R. The number of forwarding table entries increases with the increasing number of IXP participants. Each of our techniques progressively improves scalability. We observe that the number of forwarding table entries for iSDX-R is very close to the lower bound (i.e., best case), where the number of forwarding table entries is equal to the number of SDN policies (for each participant, on average 2.5 policies towards 10% of all participants).

We also explore the effects of distributing the control plane computation on the ability of iSDX to perform MDS compression. The results are shown in Figure 9. Given 500 participants, distributing the control plane reduces the number of next hop entries for the border router from 25,000 to 360. This reduction mitigates the load on the border routers, since the number of virtual next hop IP addresses reflects the number of ARP entries each participant’s border router must maintain.

**Time to perform policy compression.** Figure 10 shows the compression time for each controller; this time dominates control-plane computation but only occurs at initialization. The Centralized MDS-SDX operates on a large input rule matrix, and thus requires nearly five minutes to compress policies. iSDX-D distributes the computation across participants, reducing compression time by three orders of magnitude. iSDX-R takes longer than iSDX-D and iSDX-N controllers. For 500 participants, policy compression takes about three seconds.
Figure 9: Number of virtual next-hop IP addresses for centralized and distributed control planes. Results for distributed iSDX do not depend on encoding or compression approach.

Figure 10: Time to perform policy compression.
3.2.3 Runtime Performance

After iSDX initializes, we replay a two-hour trace of BGP updates from one of the largest IXPs in the world to evaluate the runtime performance of iSDX compared to other SDX designs. We focus on how iSDX reduces the number of forwarding table updates induced by BGP updates and policy changes, as well as the corresponding increase in gratuitous ARP traffic, which is the cost we pay for increased forwarding table stability.

**Forwarding table updates in response to routing.** Figure 11 shows the cumulative distribution of the number of updated forwarding table entries per second the SDX must process for a BGP update stream coming from all 511 participants at the IXP. MDS compression, which is used in iSDX-D and iSDX-N, significantly increases the rate of updates to the forwarding table in comparison to an unoptimized SDX; this result makes sense because any change to forwarding is more likely to trigger a change to one of the encoded forwarding table entries. With iSDX-R, there are never updates to the forwarding table entries in response to BGP updates.

**Update latency in response to BGP updates.** We aim to understand how quickly iSDX-R can update forwarding information when BGP updates...
arrive. For iSDX-R, this update time effectively amounts to computing updated virtual next-hop IP and MAC addresses, since iSDX-R never needs to update the IXP fabric forwarding table entries in response to BGP updates. We evaluate update latency with two experiments. First, we vary the fraction of IXP participants to which each IXP participant forwards with
Figure 14: Rate at which a participant’s border router receives gratuitous ARPs.

SDN policies. For example, if the fraction is 1, each participant has between one and four SDN forwarding policies (at random) for every other SDN participant. Figure 12 shows this result; in all cases, the update latency in response to a BGP update is less than ten milliseconds, and the 95th percentile in the worst case is less than 20 ms. Even when we perform simultaneous compilation of all 511 participants on just three servers at the IXP, the median update time is only 52 ms, well within practical requirements.

To understand how iSDX-R behaves when it receives larger update bursts, we evaluate the update latency for increasing sizes of BGP update bursts. We vary the number of BGP updates per second from 20 to 100 and send a constant stream of updates at this rate for five minutes, tracking the latency that the iSDX requires to process the updates. (Although a table reset would presumably cause a very large update burst, the fastest sustained BGP update rate we observed in the trace was only about 35 BGP updates per second.) Figure 13 shows this result. For example, for a rate of 100 BGP updates per second, the median update latency is about 8 ms and the 95th percentile is percentile is about 45 ms.
Gratuitous ARP overhead. Recall that SDX relies on gratuitous ARP to update virtual destination MAC addresses when forwarding behavior changes, often in lieu of updating the forwarding table itself. A centralized SDX control plane sends this ARP response to all IXP participants, but a distributed SDX can send this response only to the border router whose route changed. Figure 14 shows the distribution of the rate at which a participant’s border router receives gratuitous ARP messages from the IXP controller in response to BGP routing changes, for both the centralized design (i.e., centralized MDS) and the distributed one (i.e., iSDX); these rates are independent of which encoding the iSDX uses.

3.3 Core Switches

In this section we introduce our forwarding mechanism used within the IXP network, which we call Umbrella. Its goal is to enhance the scalability and stability of legacy IXP fabrics, taking advantage of the SDN paradigm. We show that Umbrella scales on existing hardware, and does not require a central point of control.

3.3.1 No More Broadcast Traffic

IXPs apply strict rules to limit the side effects of a layer-2 shared broadcast domain, e.g., the router MAC address of the member that connects to the peering fabric must be known in advance. Only then the IXP will allocate an ethernet port on the edge switch as well as an IP address from the peering IXP IP Public Space and configure a MAC filtering ACL with that MAC address. As a consequence, the location of all member’s routers is known to the IXP and does not change as dynamically as assumed by the layer-2 protocols. Furthermore, we exploit the ability of OpenFlow to rewrite the destination MAC address of a frame matching a given rule, enabling on-the-fly translation of broadcast packets into unicast at the edge of the fabric. This can be exploited to eliminate location discovery mechanisms based on broadcast packets (i.e., ARP request, IPv6 neighbor discovery). Figure 15 shows a high-level vision of the proposed idea using Figure 16 as a reference scenario.

We propose a label-oriented forwarding mechanism to reduce the number of rules at core level. In particular, Umbrella edge switches explicitly write the per-hop port destination into the destination MAC field of the packet. The first byte of the MAC address represents the output port the core switch has to use.
Table 2 shows an example on how a Core Switch flow table looks like in the Umbrella architecture. While this encoding scheme is currently limited to 256 output ports per hop, more bits in the port encoding (thus mapping more physical ports) can be used. With Umbrella, the number of flow table entries per core switch will therefore scale with the number of active physical ports in the switch itself. This aspect is important to tackle the address resolution problem directly from within the data plane.

Let’s take as example the topology shown in Figure 16 and consider the case where edge-3 is connected to a core switch through port number 2.
and to router-b through port number 3. Finally, let’s take the case where router-a sends an ARP request (i.e., broadcast message) to router-b. Edge-1 receives the frame, rewrites the destination MAC address using the following encoding: 02:03:00:00:00:00 and forwards it to the right core switch. Once the frame reaches the core, it is redirected to output port 2 (i.e., the forwarding in the core is based on the most significant byte) to switch edge-3. Finally, edge-3, before forwarding the frame through the output port indicated in the second byte of the MAC address, rewrites that field with the real MAC address of router-b. In case the source and destination are directly connected to the same edge switch, no encoding is needed, and the broadcast destination address is directly replaced by the target MAC destination address by the edge switch. In an IPv6 scenario, the OpenFlow match pattern indicated in the edge switch needs to be on the IPv6 ND target field of the incoming ICMPv6 Neighbor Solicitation packet [30]. The matching table on the edge switch should maintain an association between IPv6 addresses and their location, as in the IPv4 case.
3.3.2 Towards a Segment Routing-Like Approach

The forwarding mechanism we proposed so far has the advantage of allowing to reuse legacy switches in the core, limiting the burden (and costs) to upgrade an IXP fabric to the Umbrella architecture. In this scenario, a core switch only needs to forward packets based on simple access filtering rules, while the edge switches need OF-like capabilities to rewrite the layer-2 destination field.

While this approach is directly applicable to fabrics that rely on a single hop in the core (as AMS-IX and DE-CIX do), it is not applicable to fabrics with multiple hops (as LINX and MSK-IX do). With a single hop, the core switch would expect the output port to be encoded in the most significant byte of the destination MAC address. In the multi-hop case on the other hand, since a packet can traverse multiple core switches, a new encoding scheme is needed to differentiate the output ports at different core switches.

![Figure 17: Example of multi-hop in the core.](image)

Figure 17 shows an example with multiple hops in the core. To reach edge-d, edge-a needs to cross two different core switches through path b. This is a fairly common case in hypercube-like topologies, as the ones adopted by LINX or MSK-IX. In this scenario, following the Umbrella approach, it is straightforward to propose an encoding of the layer-2 destination address where the most significant byte refers to the output port of the first core switch (i.e., core-a), the second byte to the second
switch (i.e., core-b), and so on. Unfortunately, depending on the actual route being used, a core switch might be the first or the second on the path, making the proposed approach unfeasible. Another solution is to take into account also the input port of the frame in the forwarding rules installed in the core switches. Given the input port, it is possible to know where the switch is on the path and therefore look at the right byte in the layer-2 destination address. Unfortunately, this approach may not work in arbitrary topologies neither. Moreover, it will experience a rule explosion in the core, i.e., the number of forwarding entries grows quadratically with the number of possible input ports, making such an idea not very attractive.

These problems can be addressed using a segment routing-like approach. Segment Routing leverages the source routing paradigm, keeping the Umbrella spirit, where the first edge switch is in charge of selecting the path. Segment Routing consists in each node steering a packet through an ordered list of instructions, called segments, in this case the output ports. An ordered list of segments is encoded as a stack of labels. The segment to be processed is on the top of the stack, and popped upon completion of a segment.

When a new frame reaches the fabric, it has to pass through a first edge switch in charge of rewriting the MAC destination address with an ordered list of output ports. Each port refers to a different core switch on the path towards the destination. When a core switch receives the frame, it looks up the most significant byte of the address to get the destination output port and rewrites the address by shifting the value to make the second byte the new most significant one (Figure 18).

![Figure 18: Example of Umbrella forwarding scheme.](image)

Each switch needs only to look at the most significant byte of the address,
no matter where it is on the path toward the destination. After the lookup, the address must also be rewritten, making this solution feasible only when OpenFlow-enabled switches are used in the core. Every core switch must have 2 action tables: forwarding and copy-field. This solution comes with two main practical limitations (which will be addressed in Section 3.4):

- The maximum number of output ports that can be addressed per-hop is 256, as we embed the output port for each core switch in the most significant byte of the layer-2 destination address.

- The maximum number of hops inside the IXP must be less than 6, as we use the 6 bytes of the MAC address to encode the whole path of the frame.

Despite these practical considerations, Umbrella can actually be seen more as a generic approach for IXP fabric operation, not just an ad-hoc solution to specific layer-2 issues. We believe that the general concept of Umbrella is its main strength, i.e., offloading the control plane with a more intelligent data plane, as we illustrate with the following example of BGP.

### 3.3.3 Umbrella Interaction with Route Servers

Initially, if two IXP member ASes wanted to exchange traffic through the IXP’s switching fabric, they had to establish a bi-lateral (BL) BGP peering session at the IXP. Each BGP node connected to the fabric who wanted to obtain information about all reachable networks prefixes at the IXP had to establish BGP peerings using TCP connections with other IXP members. As IXPs grew in size (Figure 19), this solution proved impractical, as that implied keeping too many BGP sessions, with the associated administrative overhead, operational burden, and risks to push IXP members router hardware to its limits. IXPs introduced Route Servers as an answer, and offered them as a free value-added service to their members [36]. Route Servers store all the incoming route information from IXP members and forward them without modification. Thanks to the RS, an IXP member can receive all the routing information available to the IXP through a single BGP session.

Umbrella can be used to handle the forwarding of BGP traffic within an IXP, both for usual bi-lateral peerings and RSs. For bi-lateral BGP sessions, the TCP connection is treated as pure data plane traffic crossing the IXP, and the traffic is handled by the switch rules, without control plane intervention. For RSs, the BGP traffic entering the fabric is directed to the
RS thanks to a single rule at the edge switch, while the egress traffic is handled automatically through the existing rules on the edge switches. This example illustrates how simple the handling of specific control plane traffic can be with Umbrella, limiting the growth of the number of rules inside the IXP fabric.

3.4 Preliminary Simulations at the Core Switches

In this section, we evaluate several aspects of the Umbrella architecture. We first estimate the average number of flow rules needed at the edge, taking into account different IXP fabric sizes (Section 3.4). We then discuss the applicability of the segment routing approach in real scenarios (Section 3.4). Finally, we compare the impact of the ARP-induced broadcast storm over BGP connections between a legacy scenario and an Umbrella-enabled one (Section 3.4).

Flow table occupancy at the Edge Mapping the broadcast destination MAC address into the path inside the fabric requires additional flow rules at the edge of the fabric. This section provides a scalability analysis in
terms of required number of rules. From an ingress traffic point of view, the proposed architecture requires the OpenFlow switches at the edge to store three entries per peering router connected to the fabric. This is a necessary and sufficient condition to enable routing in any situation (i.e., IPv4 ARP, IPv6 Neighbor Solicitation and dataplane traffic). In the following we show a sample of the three entries referred to a given peering router:

- "eth.type":0x806, "arp_op":1, "arp_tpa":"1.1.1.3", "actions":["type":"SET_FIELD", "field":"eth_dst", "value":"18:01:00:00:00:00", "type":"OUTPUT", "port":"49"]
- "eth.type": 0x86DD, "ip_proto":58, "icmpv6_type":135, "ipv6_nd_target":"2001::1/128", "actions":["type":"SET_FIELD", "field":"eth_dst", "value":"18:01:00:00:00:00", "type":"OUTPUT", "port":"49"]
- "dl_dst":"00:00:00:00:00:03", "actions":["type":"SET_FIELD", "field":"eth_dst", "value":"18:01:00:00:00:00", "type":"OUTPUT", "port":"49"]

The first entry handles the ingress IPv4 ARP traffic. It is important to lookup for the IP address of the peering router towards which the location discovery traffic is targeted. The destination MAC address field (i.e., broadcast) will be rewritten with a pre-defined path through the fabric. The second and third entries refer to IPv6 Neighbor Solicitation and dataplane traffic. Note that all the three entries share a common action in the rewriting and forwarding process, because Umbrella processes location discovery and dataplane traffic in the same way. From an egress traffic point of view, each edge switch needs to rewrite the MAC destination field of the receiving frame with the correct target. As the value to be inserted in the destination MAC field depends on the output port of the edge switch, the number of rules here also depends on the number of peering routers connected to the edge switch. The total number of rules for an edge switch is then the sum of the ingress and the egress ones.

Figure 20 shows the average number of rules needed per edge switch if the Umbrella architecture was to be used in different European exchange points.
As these numbers depend on the number of peering routers connected to a fabric, we conducted an analysis on PeeringDB [3,22].

Given the number of flow entries required as shown on Figure 20, it appears that the Umbrella architecture is applicable in today’s IXP. Indeed, today’s OpenFlow-enabled switches can already support from few thousands flows (e.g., Pica8 switches\(^5\)) to hundreds of thousands of flows (e.g., Corsa\(^6\) and Noviflow\(^7\) switches).

**Maximum number of hops through IXP fabric** We already mentioned one of the main limitations of our approach in Section 3.3.2: the maximum number of hops inside the IXP must be less of 6, due to our use of the destination MAC address field to embed at the edge the path of the frame (8 bit per hop). While this requirement is typically met by design in current fabrics which allow only one hop in the core, we feel that it might not be the case when multi-hop in the core is permitted, especially

---


\(^6\) [http://www.corsa.com/sdn-done-right/](http://www.corsa.com/sdn-done-right/)

\(^7\) [http://noviflow.com](http://noviflow.com)
in backup/recovery after some link failure, where routing inside the fabric may be allowed to follow longer paths. We therefore studied the publicly available topologies of four large European IXP fabrics, to get an idea of the number of hops a packet must traverse at the exchange in the worst case.

![Graph showing average number of hops in large European exchange points.](image)

Figure 21: Average number of hops in large European exchange points.

Figure 21 shows a per fabric maximum number of hops in case of normal operation and fallback due to massive link failures. We define massive link failure as the maximum number of link failures that the fabric can sustain while still guaranteeing connectivity between each peering router connected to the fabric. The results shown during normal operation for the AMS-IX and DE-CIX fabrics are expected, since they allow only one hop in the core. The main difference between the two fabrics comes in case of massive link failure. While AMS-IX duplicates the core switches (leading to a maximum of three hops), DE-CIX uses four core switches connected in a full mesh topology (leading to a maximum of four hops). Although our requirement is not met in the case of LINX in the fallback/recovery situation, we believe that this is actually not a real limitation of the proposed solution, as such failures would disrupt significantly the fabric anyway. The LINX topology can be seen as a hypercube where each vertex is a core switch. The maximum
The number of hops during a massive link failure situation can therefore be as high as 9. If necessary, such situations should be handled through protection mechanisms, e.g., pre-computed MPLS paths, as the traffic pattern through the remaining part of the fabric will have to be carefully engineered.

**Effects of the ARP-induced broadcast storm**  We emulated a general IXP topology using Mininext [42]. The test topology created was composed by eight edges switches connected by two cores (Open vSwitch instances – OvS). The edges were attached to Mininext container hosts running Quagga and configured as BGP peering routers. Moreover, a physical route server and a real router (i.e., Cisco c3640), were also connected to the proposed topology through the interfaces of the server running Mininext. This is an important aspect as we want to quantify the impact against a real device, not in an emulated environment. The first test verified the ARP-induced broadcast storm effect on a peering router for a small IXP fabric. To study its impact over BGP connections in a legacy IXP scenario, we used OvSs as traditional layer-2 switches configured with the Spanning Tree Protocol (STP). We included a hundred Mininext nodes peering with the route server and the Cisco router.

Figure 22 shows the evolution of the average CPU occupancy on the Cisco router before, during and after the broadcast storm. The grey zone represents the time span during which the router looses BGP connectivity with the route server. Before the ARP explosion, the average CPU occupancy is low, showing only a peak at the beginning, when the ARP resolution process starts. When the hosts start to generate ARP messages for non-reachable addresses (after minute 6), the packets reach the Cisco device, bringing its average CPU occupancy close to 100%. Afterwards, the router BGP session with the route server goes down and returns back only when the broadcast storm ends.

Figure 23 shows the evolution of the Cisco router BGP state machine during the test. Each block shows the message exchange between the router and the route server given the current BGP state. After minute 3, the router shows an average CPU occupancy higher than 90%, making it unable from processing the route server KEEPALIVE messages. As the router BGP hold timer expires, a NOTIFICATION message is released and BGP session goes down. The BGP state becomes Idle and then Connecting as the router tries to re-initiate the TCP connection. Unfortunately, the high CPU load prevents the router from processing the route server SYN-ACK messages (as shown in the Active state block). It is clear the impact of the CPU load.
Figure 22: Effect of an ARP-induced broadcast storm on a peering router CPU occupancy.

on the overall performance of the fabric.

Figure 24 compares the average CPU load on the Cisco router between (1) a legacy, (2) a legacy with the ARP-sponge server support and (3) an Umbrella enhanced scenario under broadcast storm event. We used a topology with 20 peering routers and instruct a server to act as ARP-sponge. The size of the emulated fabric is determined by the resources available in the server running Mininet. The legacy scenario experienced a CPU increase up to 100% on average (leading to a BGP failure), while the one supported by the ARP-sponge shows an average CPU occupancy of 50% with peaks of 90% (the server does not sponge automatically but has a pending value). As this is a small fabric, and ARP-sponge already barely reduces the impact of the storm, it is clear that ARP-sponge cannot keep pace with ARP storms in medium to large fabrics. The Umbrella scenario on the other hand is not affected by the event, keeping the BGP connection alive, regardless of the fabric size.

8The ARP-sponge code is available at the following link: http://ams-ix.net/downloads/arpsponge/
4 Reliable Routing

In this section, we explain how ez-Segway provides routing reliability during network routing updates. We first describe the update activities that are performed while managing a network and then we formalize the routing properties that we want to guarantee during a routing transition to a new state. The network consists of switches \( S = \{ s_i \} \) and directed links \( L = \{ \ell_{i,j} \} \), in which \( \ell_{i,j} \) connects \( s_i \) to \( s_j \) and has capacity \( \ell_{i,j}.cap \).

**Flow.** A flow \( F \) is an aggregate of packets between an ingress switch and an egress switch. Every flow is associated with a traffic volume \( v_F \). In practice, this volume could be an estimate that the controller computes by periodically gathering switch statistics \cite{14} or based on an allocation of bandwidth resources \cite{13}. The forwarding state of a flow consists of an exact match rule that matches all the packets of the flow. As in previous work \cite{14}, we assume that flows can be split among multiple paths by means of weighted load balancing schemes such as WCMP.

**Network configuration.** A network configuration \( \mathcal{C} \) is the collection of all forwarding states that determine what packets are forwarded between any pair of switches and how they are forwarded.
Figure 24: Comparison between the CPU occupancy of a peering router connected to a legacy fabric, a legacy fabric with the ARP-sponge server activated and the Umbrella fabric for a small exchange point.

4.1 The Network Update Problem

Given two network configurations $C, C'$, a network update is a process that replaces the current network configuration $C$ by the target one $C'$.

We focus on four consistency properties of network update:

i) black-hole freedom: For any flow, no packet is unintendedly dropped in the network.

ii) loop-freedom: No packet should loop in the network.

iii) congestion-freedom: No link has to carry a traffic greater than its capacity.

iv) endpoint-policy coherence: Every packet should be forwarded according to either the old or new endpoint policy but never a mixture of the two.

From these properties, the first three are the same as the ones studied in [25]. The fourth one is the relaxation of the per-packet consistency [35].

H2020-ICT-2014-1 Project No. 644960
endpoint policy specifies what to forward but not how — that is, it ignores the internal topology details as though the network was one big virtual switch [15,18]. To achieve this property, we posit every packet carries a tag identifying the version of the endpoint policy applied to the packet.

Due to the limited capacity of physical links and the inherent difficulty in synchronizing the changes at different ingress switches, the link load during an update could get significantly higher than that before or after the update and all flows of a configuration cannot be moved at the same time. Thus, to minimize disruptions, it is necessary to decompose a network update into a set of update operations \( \Pi \). An update operation \( \pi \) denotes the operation necessary to move a flow \( F \) from the old to the new configuration: in the context of a single switch, this refers to the addition or deletion of \( F \)’s forwarding state. At anytime, only some update operation in a subset of \( \Pi \) is executed. That leads the network to an intermediate configuration. The update is successful when the network is transformed from the current configuration \( C \) to the target configuration \( C' \) such that for all intermediate configurations, four consistency properties are preserved.

4.2 The Idea

Our goal is to perform fast network updates while preserving desirable consistency properties (no loops, no blackholes, packet coherence) and avoiding congestion.

Prior approaches to the network update problem have ranged from preserving strong consistency properties [6, 26, 28, 35] to avoiding congestion [19], reducing space overheads [16], relaxing consistency properties [23, 25], or adapting update plans based on dynamic conditions [14,34].

All these approaches have used a central controller to plan and coordinate the update. As a thought experiment, we depart from this common assumption and ask: could the network update problem be solved in a decentralized fashion wherein switches are delegated the task of implementing the update?

We now show a series of examples to overview our approach and illustrate the challenges it must overcome. We use the simple network of seven switches \( s_1, \cdots, s_7 \) shown in Figure 25. Assume each link has 10 units of capacity. The network forwards traffic aggregates shown as differently color-coded flows \( F_1, \cdots, F_n \), each of a given size (annotated).
4.2.1 Decentralizing for Fast Consistent Updates

Consider the example in Figure 25 where the network configuration needs to be updated from $C_1$ to $C_1'$. The size of each flow is 5; so, every link can carry at most 2 flows at the same time.

Note that we cannot simply transition to $C_1$ by updating all switches at the same time. Since switches will apply updates at different time, such strategy can neither ensure that links do not become congested during the update nor that the network remains free of loops and blackholes. For example, if $s_2$ forwards $F_1$ on link $\ell_{2,6}$ before $F_3$ is updated, then that link would become congested. Moreover, if $s_3$ is updated to forward $F_3$ on link $\ell_{4,7}$ before the forwarding state for $F_3$ is installed at $s_7$, this results in a temporary blackhole.

Ensuring that the network stays congestion free and that the consistency of the forwarding state is not violated requires us to carefully plan the order of update operations across the switches. In particular, we observe that certain operations depend on other operations, leading potentially to long chains of dependencies for non-trivial updates. This fact implies that a network update can be slowed down by two critical factors: (i) the amount of computation necessary to find an appropriate update schedule, and (ii) the inherent latencies affecting communication between controller and switches summed over multiple rounds of communications to respect operation dependencies.

But is it necessary to have the controller involved at every step of a network update? We find that it is possible to achieve fast, consistent updates by minimizing controller involvement and delegating to the switch the tasks of scheduling and coordinating the update process.

We leverage two main insights to achieve fast, consistent updates. Our
first insight is that we can complete an update faster by using in-band messaging between switches instead of coordinating the update at the controller, which pays the costs of higher latency. Our second insight is that it is not always necessary to move a flow as a whole. We can complete the update faster by using segmentation, wherein different segments of a flow can be updated in parallel. For example, in Figure 25 both flows $F_1$ and $F_3$ have one segment between $s_2$ and $s_3$, and one between $s_3$ and $s_4$.

Segments can be safely updated independently and in parallel while still ensuring loop-, black-hole-, and congestion freedom. Segmentation relaxes per-packet consistency but preserves the same endpoint policy.

**Example.** Before we introduce our ez-Segway approach in detail, we illustrate the execution of a decentralized network update for the example of Figure 25. Initially, the controller sends to every switch a message containing the current configuration $C_1$ and target configuration $C'_1$. This information allows every switch to compute what forwarding state to update (by knowing which flows traverse it and their sizes) as well as when each state update should occur (by obeying operation dependencies while coordinating with other switches via in-band messaging).

In the example, switch $s_2$ infers that link $\ell_{2,3}$ has enough capacity to carry $F_3$ and that its new next hop switch, $s_3$, is already capable of forwarding $F_3$ (because the flow traverses it in both old and new configurations). Therefore, $s_2$ updates its forwarding state so as to move $F_3$ from $s_2 \rightarrow s_6 \rightarrow s_3$ to $s_2 \rightarrow s_3$. It then notifies $s_6$ about the update of $F_3$, allowing $s_6$ to safely remove the forwarding state corresponding to this flow. Similarly, $s_3$ updates its forwarding state for $F_1$ to flow on $\ell_{3,4}$, notifying $s_7$ about the update. Meanwhile, switch $s_7$ installs forwarding state for $F_3$ and notifies $s_3$. Then $s_3$ moves $F_3$ onto $\ell_{3,7}$.

Once notified by $s_2$, $s_6$ infers that link $\ell_{6,3}$ now has available capacity to carry $F_1$. So, $s_6$ installs the corresponding forwarding state and notifies $s_2$, which is the upstream switch on the new path of $F_1$. Then, $s_2$ infers that the new downstream path is ready and that $\ell_{2,5}$ also has enough capacity, and it moves $F_1$ to its new path.

Notice that several update operations can run in parallel at multiple switches. However, whenever operations have unsatisfied dependencies, switches must coordinate. In this example, the longest dependency chain involves the three operations that must occur in sequence at $s_2$, $s_6$, and $s_2$ again. Therefore, the above strategy accumulates the delay for the initial message from the controller to arrive at the switches plus a round-trip delay between $s_2$ and $s_6$. In contrast, if a centralized approach performed the
4.2.2 Dealing with Deadlocks

As we consider a decentralized approach, it is worth asking: is it always possible to reach the target configuration while preserving all the desired consistency properties and resource constraints during the transition? Unfortunately, as in the centralized case, some updates are not feasible: that is, even if the target configuration is valid, there exists no ordering of update operations to reach the target. For example, consider Figure 26: if we first move $F_1$, link $\ell_{2,6}$ becomes congested; if we first move $F_3$, link $\ell_{2,3}$ becomes congested.

Assume that a feasible update ordering exists. Then, will a decentralized approach always be able to find it? Unfortunately, without computing a possible schedule in advance [26], inappropriate ordering can lead to deadlocks where no further progress can be made. However, as discussed in [14], computing a feasible schedule is a hard problem: it is NP-complete in the presence of both link capacity and switch memory constraints, and finding the fastest schedule with link capacity constraints is NP-complete as well.

In practice, even in centralized settings the current quickest approach, Dyonisus [14], cannot entirely avoid deadlocks as it uses a heuristic solution to compute a schedule that adapts to dynamic conditions. To resolve deadlocks, Dionysus reduces flow rates to continue update without violating the consistency properties; however, this comes at the cost of lower network throughput.

Since deadlocks pose an important challenge for us, and the decentralized approach is potentially more likely to enter deadlocks due to the lack of global information, we develop two new techniques that avoid reducing network throughput to deal with deadlocks. We defer the discussion on how to detect deadlocks to later in Section 4.3.2.

Our first technique is splitting volume, which divides a flow’s traffic onto its old and new paths to resolve a deadlock. The second technique is segmentation, which allows to update different flow segments independently of one another. As we discussed, segmentation allows an update to complete faster via parallelization. In this case, it also helps to resolve certain deadlocks. Before presenting our techniques in detail, we illustrate them via two intuitive examples.
Splitting volume example. Consider the example in Figure 26 where every flow has size 4. This case presents a deadlock because we cannot move $F_1$ first without congesting $\ell_{2,6}$ or move $F_3$ first without congesting $\ell_{2,3}$.

We resolve this deadlock by splitting the flows. Switch $s_2$ infers that $\ell_{2,3}$ has 2 units of capacity and starts moving the corresponding fraction of $F_3$ onto that link. This movement gives sufficient capacity to move $F_1$ to $\ell_{2,6}$. Once $F_1$ is moved, there is sufficient capacity to complete the mode of $F_3$.

Note that, we could even move 2 units of $F_1$ simultaneously to moving two units of $F_3$. This is what our decentralized solution described later would actually do.

Segmentation example. Consider now the example in Figure 27 where every flow has size 5. This case presents a deadlock that prevents flows from being updated all at once. In particular, if we first move $F_1$, link $\ell_{3,4}$ becomes congested. Similarly, if we first move $F_3$, link $\ell_{2,3}$ is then congested.

We resolve this deadlock by segmenting these flows as: $F_1 = \{s_2 \ldots s_3, s_3 \ldots s_4, s_4 \ldots s_5\}$, $F_3 = \{s_1 \ldots s_2, s_2 \ldots s_3, s_3 \ldots s_4\}$. Then,
switches $s_2$ and $s_6$ coordinate to first move segment \{s_2 \ldots s_3\} of $F_1$ followed by the same segment of $F_3$. Independently, switches $s_3$ and $s_7$ move segment \{s_3 \ldots s_4\} of $F_3$ and $F_1$, in this order.

4.3 Ez-Segway

Ez-Segway acts as a tool that network operators can use in order to improve the completion time of the process that handles the update of the network forwarding state by means of a distributed approach that prevents routing anomalies (e.g., forwarding loops), minimize congestion (i.e., links carrying too much traffic), and preserve end-point policy consistency.

Instead of coordinating and synchronizing the update of segments from a centralized entity, ez-Segway computes the information that is needed by the switches for performing a distributed update of the network that carefully tries to avoid routing anomalies and link congestion and deadlocks and send it to each switch. Once the switches complete the network update operation, the controller is notified of the outcome of the network update.

We first describe in more detail our segmentation technique and then we deal with the problem of distributively updating the forwarding state while minimizing link congestion.

4.3.1 Flow Segmentation

In this subsection, we introduce our segmentation technique, which provides two benefits: it reduces the risk of update deadlocks due to congested links by allowing a more fine-grained control of the flow update and it speeds up the migration completion time of a flow to its new path. This procedure is performed by the controller when a network state update is triggered. We first introduce the idea behind segmentation with simple illustrative examples and then provide formal definitions.

Consider the update problem represented in Figure 28a, where a flow needs to be moved from the old (black) path \(s_0 \ldots s_2s_3\) to the new (red) path \(s_0 \ldots s_2s_3\). A simple approach would work as follows. A message is sent from $s_3$ to $s_0$ along the new path for acknowledging that the new path is installed with higher priority than the old path and it is safe to switch to the new path without the risk of neither black-holes nor forwarding loops. Once $s_0$ switches to the new path, a message is sent from $s_0$ towards $s_3$ along the old path for acknowledging that no packets will be forwarded anymore along the old path, which is then safe to be removed. We call to this migration technique from an old path to the new one Basic-Migration. It is easy to
observe that Basic-Migration prevents forwarding loops and black-holes.

**Theorem 1** Basic-Migration is black-hole-free and forwarding-loop-free.

It can be observed that the whole updated operation could be performed much faster. With segmentation the subpath (s₀s₄s₁) of the old (black) path can be updated with Basic-Migration to (s₀s₆s₁) while subpath (s₁s₅s₂) of the old (black) path is updated with Basic-Migration to (s₁s₇s₂). In fact s₆ does not have to wait for s₁ to update its path since s₁ is always guaranteed to have a forwarding path towards s₃. We say that the pairs (s₀s₄s₁, s₀s₆s₁) and (s₁s₅s₂, s₁s₇s₂) are two segments of the flow update. Namely, a segment of a flow update from an old path $P_o$ to a new path $P_n$ is a pair of subpaths $P'_o$ and $P'_n$ of $P_o$ and $P_n$, respectively, that start with the same first switch. We denote a segment by $(P'_o, P'_n)$. The action of updating a segment consists in performing Basic-Migration from $P'_o$ to $P'_n$.

In some cases, finding a set of segments that can be all updated in parallel is not possible. For instance, in Figure 28b, we need to guarantee that s₂ updates its next-hop switch only after s₁ has updated its own next-hop, otherwise a forwarding loop along (s₂, s₁, s₅) arises. While one option would be to update the flow using Basic-Migration along the reversed new path, some parallelization is actually still possible. In fact, we could update segments $S_1 = (s₀s₄s₁, s₀s₆s₂)$ and $S_2 = (s₁s₅s₂, s₁s₇s₃)$ in parallel without any risk of creating a forwarding loop. In fact, s₂, which is a critical switch, is not yet updated and the risk of forwarding oscillation is avoided. After s₁ is update, also segment $S_3 = (s₂s₆s₃, s₂s₈)$...
can be updated. Every time two switches appear in reversed order in the old and new path, one of the two switches have to wait until the other switch complete its update.

**Anomaly-free updates with segmentation.** Computing a set of segment that maximize the number of segments while minimizing the chain of dependencies among them is not an easy task. We rely on an heuristic that works by constructing two sequences of segments $T_{\text{non-loop}}$ and $T_{\text{loop}}$ such that the $i$'th segment in $T_{\text{loop}}$ must be executed after the $i$'th segment in $T_{\text{non-loop}}$ in order to avoid forwarding loops and black-holes. We call such technique \textsc{Segmented-Migration}.

At very high-level, we identify the set of common switches $S_C$ among the old and new path and we denote by $S_R$ the pairs of switches in $S_C$ that appear in reversed order in the two paths. We select a subset $S'$ of $S_R$ that has the following properties: (i) for each two pairs of switches $(r,s)$ and $(r',s')$ neither $s'$ nor $s'$ are contained in the subpath from $r$ to $s$ in the old path, (ii) every switch in $S_R$ is contained in at least a subpath from $r$ to $s$ for a pair $(r,s)$ of $S'$, and (iii) the number of switches in $S_C \setminus S_R$ that are contained in at least a subpath from $r$ to $s$ for a pair $(r,s)$ of $S'$ is minimized. Intuitively, each pair $(s_1, s_2)$ of $S'$ represents a pair of switches that may create a forwarding loop unless $s_2$ is updated after $s_1$. The complexity for computing these pair of switches is is $O(N)$, where $N$ is the number of switches in the network. Given $S'$, we then create $T_{\text{non-loop}}$ and $T_{\text{loop}}$ from $S'$ as follows. For each pair $(r,s)$ in $S'$ we append segments $(r \ldots s, r \ldots t)$ and $(s \ldots r, s \ldots u)$ into $T_{\text{non-loop}}$ and $T_{\text{loop}}$, respectively, where $t$ ($u$) is the first switch in $S'$ that succeeds $r$ ($s$) in the new path. Finally, for any pair $(r,s)$ of switches in $S_C \setminus S'$ such that $s$ is the first switch in $S_C$ that succeeds $r$ in the new path, we append segment $(r \ldots u, r \ldots s)$ into $T_{\text{non-loop}}$, where $u$ is the first switch that succeeds $r$ in the old path. The following theorem (proof omitted) guarantees that \textsc{Segmented-Migration} does not create black-holes or forwarding loops.

**Theorem 2** \textsc{Segmented-Migration} is black-hole-free and forwarding-loop-free.

As an example, consider the flow update depicted in Figure 28c. Let $S'$ be $\{(s_1, s_3), (s_4, s_5)\}$ for which both properties (i) and (ii) holds. The resulting $T_{\text{non-loop}}$ is $< (s_1 s_3 s_3 s_5, s_1 s_5 s_5), (s_4 s_5 s_5, s_4 s_6), (s_0 s_1, s_0 s_7 s_3) >$ while $T_{\text{loop}}$ is $< (s_3 s_4, s_3 s_2 s_1), (s_5 s_6, s_5 s_1 0 s_7) >$. Observe that the last segment in $T_{\text{non-loop}}$ is not enforcing any dependency on any segment in $T_{\text{loop}}$. 

H2020-ICT-2014-1 Project No. 644960
4.3.2 Scheduling Updates

Performing a network update in a distributed manner requires reliable coordination among the switches. As we already showed in Section 4.3.1, segments’ updates that can cause forwarding loops must be executed according to a partial ordering of all the update operations. In this subsection, we consider the problem of performing a network update without congesting any link in the network. The main challenge of this operation is the ability of not reaching a deadlock state in which any segment update would create some congestion. We propose a heuristic that effectively reduces the risk of deadlocks that we positively assessed in our simulation section.

Figure 29 shows a case in which an inappropriate schedule leads to a deadlock situation. $F_1$ and $F_3$ have size 4 while $F_2$ and $F_4$ have size 3. The only updates that can be performed without congesting a link are $\pi_1$ and $\pi_2$. Which one to choose first? Choosing $\pi_2$ disables $\pi_1$, and gives only 3 units of free capacity to $\ell_{2,3}$. This is not enough to enable $\pi_3$, which leads to a deadlock that can only be solved via splitting mechanisms. Instead, choosing $\pi_1$ disables $\pi_2$ but gives 4 units of free capacity to $\ell_{2,3}$, which enables $\pi_3$. In turns, executing $\pi_3$ enables $\pi_2$, and the update can complete successfully.

Splitting volumes is less preferable than finding a clean update sequence that do not require to split flow aggregates of traffic for two main reasons: (i) splitting flows requires more migration steps and (ii) the physical hardware may lack support for splitting flows based on arbitrary weights. As such, we stress the fact that ez-Segway can be configured to disable flow splitting during network updates. We therefore have two types of deadlocks in ez-Segway. If an update migration reaches a state in which the migration cannot make any progress unless by splitting a flow, we say that there is splittable deadlock in the system. Otherwise, if even splitting flows cannot make any progress, we say that there is an unsplittable deadlock in the system.

Even from a centralized perspective, computing a congestion-free migration is not an easy task for both splittable and unsplittable techniques. Our system centrally computes a dependency graph between links and flow updates in order to guide the update migration towards a congestion-free solution that does not get stuck in a deadlock. The dependency graph is transmitted to each switch, which uses it to prioritize some flow updates over some other updates.

**The dependency graph.** The dependency graph captures dependencies but still leaves scheduling flexibility since multiple operations could be
enabled at the same time. However, any dynamic scheduling might lead to a deadlock. Potential deadlocks arise as cycles in the dependency graph.

Given a pair of current and target configurations \( C, C' \), any operation \( \pi \) depends on the availability of capacity in the new path, while it increases the corresponding available capacity in the old path. These dependencies can be represented with a dependency graph — a bipartite graph \( \mathcal{G}(\Pi, L, E_{\text{free}}, E_{\text{req}}) \), where the two subsets of vertices represent the update operation set \( \Pi \) and link set \( L \). The value associated to a link vertex \( \ell_{i,j} \) represents the current available capacity of \( \ell_{i,j} \). The two subsets of edges \( E_{\text{free}} \) and \( E_{\text{req}} \) indicate the following:

- \( E_{\text{free}} \) is the set of directed edges from \( \Pi \) to \( L \). A weighted edge \( e_{\text{free}} \) from \( \pi_p \) to a link \( \ell_{i,j} \) represents the increase in available capacity at \( \ell_{i,j} \) by performing \( \pi_p \).
- \( E_{\text{req}} \) is the set of directed edges from \( L \) to \( \Pi \). A weighted edge \( e_{\text{req}} \) from link \( \ell_{i,j} \) to \( \pi_p \) represents the available capacity of \( \ell_{i,j} \) that is necessary to enable \( \pi_p \).

Consider the example update of Figure 29(b) and (d), where flows \( F_1, F_2, \) and \( F_3 \) change paths. The corresponding dependency graph is shown in Figure 29(c). In the graph, circular nodes denote operations and rectangular nodes denote link capacities. \( E_{\text{free}} \) edges are shown in black and \( E_{\text{req}} \) edges are shown in red; the weight annotations reflect the amount of increased and requested available capacity, respectively. For instance, \( \pi_1 \) requires 4 units of capacity from \( \ell_{2,6} \) and \( \ell_{6,3} \), while it increases available capacity of \( \ell_{2,3} \) by the same units.

**Deadlock-freeness heuristic based on critical cycles.** We now introduce our key structure on which our heuristic relies for choosing which updates are more important to perform than others. A cycle in the dependency graph is critical if there exists an update not in the cycle whose execution would cause any update in the cycle to be non-executable even if other updates release some capacity to links in the dependency graph.

We assign low priority to all the updates that do not belong to any cycle in the dependency graph. These updates do not release any useful capacity for other updates. We assign medium priority to all the remaining updates that belong only to non-critical cycles. We finally assign higher priority to all the updates that belong to at least one critical cycles. Updates with lower priority should not be executed before updates with higher priority. A Breadth-First-Search (BFS) tree rooted at each update operation is performed in order to determine if an update would create a critical cycle.
To avoid deadlocks during the update, when a switch has multiple enabled operations, the switch gives priority to the update operation that is the root of a critical cycle. In addition to the priority values of each flow, the updates must satisfy the constraints imposed by \textsc{Segmented-Migration}, if there are any.

Therefore, the worst case complexity for identifying a critical cycle for a given update operation is $O(|\Pi| + |L| + |\Pi| \times |L|) \approx O(|\Pi| \times |L|)$. Consequently, for all update operations the complexity is $O(|\Pi|^2 \times |L|)$. While this might seem expensive, we note that in our decentralized setting, the computation is partitioned across switches since every switch considers just the subset of operations that depend on any outgoing link at that switch. In contrast, in a centralized setting, the controller would need to perform this computation for the entire network.

**Splittable deadlock.** Assume that in our example we execute the update operation $\pi_2$ before $\pi_1$. After $F_2$ is moved to the new path, the dependency graph becomes the one in Figure 30a. In this case, every link has 3 units of
capacity but it is impossible to continue the update. However, if we allow the traffic of flow $F_1$ and $F_3$ to be carried in both the old path and the new path at the same time, we can move 3 units of $F_1$ and $F_3$ to the new path and continue the update that enables updating the remaining part of flows. But the deadlock is not always splittable when the capacity of both relevant links is zero as shown in Figure 30b.

In the presence of a splittable deadlock, there exists a splittable flow $F_p$ and there is a switch $s$ in the new segment of $F_p$. Switch $s$ detects the deadlock and determines the amount of $F_p$’s volume that can be split onto the new segment. This is taken as the minimum of the available capacity on the $s$’s outgoing link and the necessary free capacity for the link in the dependency cycle to enable another update operation at $s$.

**Deadlocks solvable by segmentation.** Coming back to the example given in Section 4.2 whose dependency graph is shown in Fig. 30b, this deadlock is unsolvable by the splitting technique. However, as discuss before, if we allow a packet to be carried in the mix of the old and the new path of the same flow, this kind of deadlock is solvable by using segmentation.

*ez-Segway* decomposes this deadlocked graph into two non-deadlocked dependency graphs as shown in Figure 31a and 31b, therefore enabling the network to be updated. *ez-Segway* considers an update operation as a movement of a segment of a path such that: (i) every switch on the path must forward the packet with the same endpoint policy; (ii) a valid path can be any combination of segments from either the old or the new configuration. Moreover, the segmentation technique of *ez-Segway* ensures that there is no inappropriate movement of path segment, which would cause a loop failure when forwarding packets.

### 4.3.3 End-Point Policy Coherence

Since the network update may contain a mix of the old and new path, it is crucial to guarantee that a packet is processed according to the old or new end-point policy. As such, we must guarantee that the new end-point policy is being installed on all the switches on both the the old and the new
path before we can start tagging packets with it. To do so, it is sufficient to exchange a message, which we call Coherent, along both the old and new path starting from the last switch of both the new and old path. The Coherent message acknowledges the installation of the new end-point policy at the switch that generates such message.

The Coherent message of can be forwarded even if a segment of a flow still has not be updated to its new path since it is only used to indicate that the new policy is now installed in the switch.

4.3.4 Dealing with Failures

In our scheme, we purposefully avoid dealing with failures in our decentralized network update scheme. While the network update process must deal with link and switch failures, we think that doing so from within the switches simply introduces unwarranted complexity.

Instead, if during an update any such failure occurs, we believe that the controller is in the best position to recompute a new global desired state and start a new update. Note that in the presence of a switch or link failure, our update process stops at some intermediate state. At this point, once the controller is notified of the failure, it will query the switches to know which update operations were performed and uses this information to reconstruct the current network state. Knowing this state enables it to compute the new desired state.

As for the messages between switches, we posit that these packets are sent with the highest priority so that they are not dropped due to congestion and that their transmission is retried if a timeout expires before an acknowledgment is received. When a message is not delivered for a maximum number of times, we behave as though the link had failed.

4.4 Preliminary Simulations

We implemented an ez-Segway prototype and evaluate it using both Mininet emulator and simulation (for large-scale evaluation). We compare ez-Segway against a controller-centric update approach, referred to as Centralized.

4.4.1 Implementation

The prototype is written as 4.5k LoC in Python. It consists of a global controller that runs centrally as a single instance, and a local controller that is instantiated on every switch. The local controller is built on top of the Ryu controller framework. The local controller connects and manipulates the
data plane state via OpenFlow whereas it sends and receives UDP messages to/from other switches. Moreover, the local controller communicates with the global controller to receive InstallUpdate messages and to send back an acknowledgment once an update completes.

4.4.2 Prototype Evaluation

We evaluate our prototype using Mininet emulator. We compare ez-Segway versus a Centralized approach that runs virtually the same algorithms executing in the switches from a central location. This approach is a close approximation of Dionysus. Centralized deviates from Dionysus in that it supports segmentation to further speed up the update time. We break down the update time by the amount of computation performed at the central controller, within the switch’s local controller and due to communication latency. We report averages and standard deviations over a thousand updates.

We measure the total update time to install updates on a set of network topologies. In particular, we use 2 real WAN topologies from B4 and Internet2 network. Without loss of generality, we assume link capacities of 1 Gbps. We place the controller at the centroid switch of the network. We compute the delay time of every link based on the geographical distance between every two switches.

Following the methods in [9], we generate all flows of a network
Figure 33: Update time (without computation) of ez-Segway versus a centralized approach for various settings.

4.4.3 Large-scale Simulations

We measure the total update time to install updates on a set of large network topologies. In particular, we use 6 real topologies annotated with link delays and weights as inferred by the RocketFuel engine [24]. We set link capacities between 1 ∼ 100 Gbps, inversely proportional to weights. We place the controller at the switch with highest degree in the network. We only model propagation delays for control traffic. We generate flows using the same approach as in our prototype experiments. We run simulations...
for a number of flows varying from 500 to 1500 and report results for 1000 flows as we observed qualitatively similar behaviors. We generate updates by simulating link failures such that they cause a certain percentage $p$ of flows to be rerouted. We ran experiments for 10%, 25%, 50%, and 75%; for brevity, we report results for 25% and 75%. For every setting of topology, flows, and failure rate, we generate 10 different pairs of old and new network configurations, and report the average update completion time and its standard deviation.

Figure 33 shows our results, which demonstrate that ez-Segway completes updates faster than the centralized approach by 24% to 49%.

4.5 Conclusions

This section explored delegating the responsibility of consistent updates to the switches. We proposed ez-Segway, a decentralized mechanism where the controller only computes the desired network configuration and switches have an active role to realize consistent network updates that satisfying four properties: black-hole freedom, loop freedom, congestion freedom, and endpoint-policy coherence. We proved our approach correct. In practice, this approach leads to improved update times, which we quantified via emulation and simulation on a range of network topologies and traffic patterns. In these experiments, our approach improved update performance by up to 64%.

5 Summary

In this document, we presented our preliminary design of an SDN-enabled IXP architecture for the ENDEAVOUR project. The architectural design decisions were driven by the requirements set forth in Deliverable 2.1, i.e.,: (i) offering to the network operators a single, coherent, global interface to interact with the underlying network that can be leveraged by the operators themselves to develop novel applications on top of its architecture, (ii) supporting arbitrarily fine-grained routing policies that can be defined by the IXP participants via an intuitive, high-level policy specification language, (iii) providing network operators with reliable network primitive functions. While i) is naturally inherited from the adoption of SDN as the network paradigm, ii) is supported by our two core and edge forwarding handlers, iSDX and Umbrella, respectively. Requirement iii) is responsibility of the ez-Segway component, which allows for updating the data-plane in a fast and anomaly-free manner. We stress the fact that the ENDEAVOUR
architecture is the first SDX architecture that can operate at the scale of the largest IXPs while supporting arbitrarily fine-grained forwarding policies.

We assessed our architecture by running some preliminary simulations against a trace from one of the largest IXPs in the world. We found that ENDEAVOUR can compile a realistic set of policies for 500 IXP participants in less than three seconds.
6 Acronyms

WAN Wide Area Network
UDP User Datagram Protocol
WCMP Weighted Cost Multi Path
BFS Breadth-First-Search
PIB Participant Information Base
VMAC Virtual MAC
FEC Forwarding Equivalence Class
GUI Graphical User Interface
MAC Media Access Control
RS Route Server
SDX Software Defined eXchange
CPU Central Processing Unit
ND Neighbor Discovery
ICMPv6 Internet Control Message Protocol version 6
RAM Random-access memory
MPLS MultiProtocol Label Switching
IPv4 Internet Protocol version 4
IPv6 Internet Protocol version 6
SDN Software Defined Networking
RIB Routing Information Base
BGP Border Gateway Protocol
IXP Internet eXchange Point
AS Autonomous System
IP  Internet Protocol
DE-CIX  German Commercial Internet Exchange
AMS-IX  Amsterdam Internet Exchange
MSK-IX  Moscow Internet Exchange
LINX  London Internet Exchange
DDoS  Distributed Denial of Service
HTTPS  Hypertext Transfer Protocol Secure
TCP  Transport Control Protocol
WAN  Wide Area Network
ARP  Address Resolution Protocol
ND  Neighbor Discovery
ACL  Access Control List
RIS  Routing Information Service
RIPE  Réseaux IP Européens
References


[34] Peter Perešini, Maciej Kuźniar, Marco Canini, and Dejan Kostić. ESPRES: Transparent SDN Update Scheduling. In HotSDN’14, Aug 2014.


[38] RIPE. Ris raw data, 2015. [Online; accessed 23-Sep-2015].


