November 2022

ADDING NEW RACKS AND RECABLING WITHOUT NETWORK DISRUPTION AND FACILITATING MIS-CABLING DETECTION

Chakradhar Kar

Avinash Reddy Yeddula

Vikas Balakrishna Dharwadkar

Sri Goli

ADDING NEW RACKS AND RECABLING WITHOUT NETWORK DISRUPTION AND FACILITATING MIS-CABLING DETECTION

AUTHORS:
Chakradhar Kar
Avinash Reddy Yeddula
Vikas Balakrishna Dharwadkar
Sri Goli

ABSTRACT

As network fabric deployments continue to grow larger and more immense, there are challenges involving the management cabling arrangements and debugging mis-cabling issues for large-scale full mesh leaf-spine networks. Additionally, the need for the automatic discovery of every component of a large fabric, which can be expanded dynamically, becomes a challenging task. Accordingly, techniques are presented herein that aid in the automatic detection of mis-cabling issues, the automatic discovery of a rack's location within a large-scale data center, and the automatic detection of the addition of a new rack to a data center, all without disrupting network traffic within a dynamically expanding fabric. Aspects of the presented techniques provide an algorithm that can automatically detect such mis-cabling issues. Further aspects of the presented techniques support a tool that may be used to detect mis-cabling (e.g., in the case of inter-communication failures among the nodes).

DETAILED DESCRIPTION

As network fabrics grow larger and more immense each day, a challenge with a large-scale full mesh leaf-spine network concerns managing a plethora of cabling arrangements and debugging mis-cabling issues. Additionally, the need for the automatic discovery of every component of a large fabric, which can be expanded dynamically, becomes a challenging task.

Accordingly, techniques are presented herein that aid in the automatic detection of mis-cabling issues, the discovery of a rack's location within a large-scale data center, and the addition of a new rack to a data center without disrupting network traffic within a dynamically expanding fabric.

As an initial matter, it will be helpful to note that a (e.g., data center) rack can be of two types – i.e., a leaf rack and a spine rack. Every rack will have two leaves and a spine rack will additionally have two spines. Every leaf in a deployment is connected to every spine – i.e., they are connected in a full mesh topology.

Within such an arrangement, the spine switches are separated into two groups and are connected to leaf switches using separate physical backbones. The backbones are not directly interconnected, and each one carries half of the fabric's throughput. Each leaf switch is connected to both backbones, and thus to every spine switch, enabling a fully non-blocking fabric. A backbone represents a physical segment of the spine and simplifies cabling. Such a backbone may be created by grooming the fiber cable connections with a passive device that is commonly known as a fiber-shuffler box (and which is often referred to as a shuffler).

Traditionally, in a network equipment vendor's application centric infrastructure data center solution, the leaves and spines are directly connected, and fabric discovery takes place though an exchange of protocol messages between the leaf switches, the spine switches, and a policy infrastructure controller.

In contrast, according to the techniques presented herein the leaves and spines are not directly connected; rather, they are connected though a fiber-shuffler.

In the narrative that follows, the techniques presented herein will be described in detail with reference to Figure 1, below. That figure depicts elements of an exemplary arrangement that is possible according to aspects of the presented techniques and which is reflective of the above discussion.

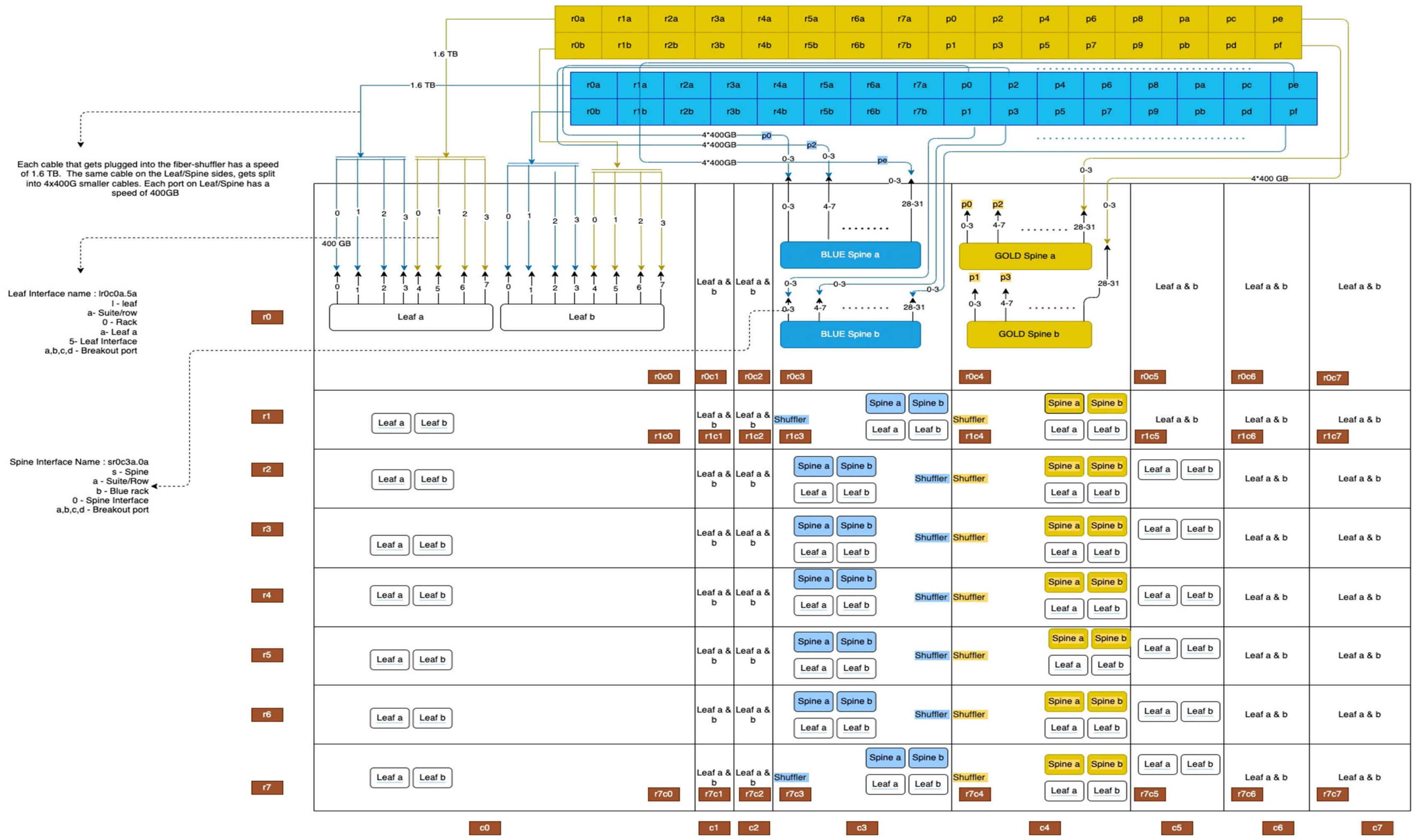

Figure 1: Exemplary Arrangement

As depicted in Figure 1, above, a fiber-shuffler is present in each and every spine rack. As a result, full mesh connectivity is achieved between the leaves and spines.

Aspects of the techniques presented herein support, among other things, the generation of a static connections database (DB) – i.e., a ‘source of truth.’ The generation of such a DB comprises a number of steps.

A first step encompasses the creation of a static list of port information (with reference to Figure 1, above, for Leafa, Leafb, Spinea, and Spineb) for a given row or rows. This is effectively configuration information that may be gathered from the connections that are depicted in a deployment figure. For example, from Figure 1, above:

```
        Blue shuffler p0 connects -> spine ports 0,1,2,3
        Blue shuffler pX connects -> spine ports *, *, *, *
        Blue shuffler r0a connects -> leaf ports 0, 1, 2, 3
        Blue shuffler leafa ports -> leafa.0, leafa.1...leafa.3
        Gold shuffler leafa ports -> leafa.4, leafa.5...leafa.7 shuffler leaf slot
indexes: r0a:0 r0b:1 r1a:2 r1b:3 r2a:4 r2b:5 r3a:6 r3b:7 r4a:8 r4b:9 r5a:10 r5b:11
r6a:12 r6b:13 r7a:14 r7b:15
```

Similarly, static lists may be created for both the blue and the gold shuffler and the corresponding switch connections.

During a second step, the leaf shuffler ports may be divided into groups. The reason for forming a group stems from the fact that a 100 Gigabit Ethernet (G) breakout link from each of the switches in the group goes to one 400G link on the spine side. Thus, every leaf connects to every spine in that row. Additionally, when new rows are dynamically added only cable swapping on the spine-side shuffler will be necessary (as will be described below) to create a full mesh connectivity.

As depicted in Figure 1, above, the group information on the shuffler (i.e., leaf) side comprises:

```
        r0a:0 r0b:0 r1a:0 r1b:0 // Group 0
        r2a:1 r2b:1 r3a:1 r3b:1 // Group 1
        r4a:2 r4b:2 r5a:2 r5b:2 // Group 2
        r6a:3 r6b:3 r7a:3 r7b:3 // Group 3
```

A third step encompasses the generation of connection information for each row independently. The following pseudo code presents one way in which such information may be developed:

```
// All of the leaves in the deployment
for every leafL in range {lr0c0a, lr0c0b, ..... lr7c7b}
  // 4 ports are connected to blue shuffler
  for every leafPort in range { 0, 1, ... 3}
    // 400G link breaks out into 4x100G
    for every breakOutIndex in range { 0, 1, 2, 3}
      // Blue shuffle connections only
      spineKey = pX where X runs from {0 - f}
      // E.g., sr0c3a.0, sr0c3a.1,sr0c3a.2, sr0c3a.3 for p0
      spineConn = blue shuffler connections for a given spineKey
      // leafShuffler side connections r0a=a, r0b=b, r1a=c, r1b =d,
      // r2a = a so on...
      spineBreakoutPort = a or b or c or d
      //LeafShuffleGroup from Step 2 (as described above)
      spineBreakoutInterface = spineConn[leafShuffleGroup]
        + spineBreakoutPort
      // breakoutIndex=0,1,2,3 -> a,b,c,d
      leafBreakOutPort = a or b or c or d
      leafBreakoutInterface = leafPort + leafBreakOutPort

      // Here we have the DB entry (2 type of DBs, leaf to spine,
      // spine to leaf)
      leafBreakoutInterface ----maps to---> spineBreakoutInterface
      spineBreakoutInterface ----maps to---> leafBreakoutInterface

  // Repeat the same steps for gold shuffler (indexing changes as shown
  // in Figure 1, above)
  // 4 ports are connected to gold shuffler
  for every leafPort in range { 4, 5, ... 7}
    // 400G link breaks out into 4x100G
    for every breakOutIndex in range { 0, 1, 2, 3}
      ....
      ....
      ....
  // Here we have the DB entry (2 type of DBs, leaf to spine,
  // spine to leaf)
  leafBreakoutInterface ----maps to---> spineBreakoutInterface
  spineBreakoutInterface ----maps to---> leafBreakoutInterface
  // connections <- append the individual entry
// All of the connections for a given leaf
connectionsDB [leaf] = connections
```

During a fourth step, once all of the individual row DBs have been generated, scripts may be written to simulate the exact actions (e.g., a cable swap steps section) that are to be performed during the addition of a new row. This results in one static connections DB containing the information for all of the connections for all of the types of deployments (e.g., one row, two rows, four rows, and eight rows). Such a static DB may be the source of truth for detecting mis-cabling and rack discovery.

A fifth step encompasses the cable swap actions that are to be performed during the addition of new rows. Two exemplary sets of actions (continuing with the arrangement that was depicted in Figure 1, above) are presented below.

For a two-row deployment, the only cable connections that must be changed upon the addition of a new suite are:

```
#1. swap row0-shuffler-blue-p2 --- row1-shuffler-blue-p0
#2. swap row0-shuffler-blue-p6 --- row1-shuffler-blue-p4
#3. swap row0-shuffler-blue-pa --- row1-shuffler-blue-p8
#4. swap row0-shuffler-blue-pe --- row1-shuffler-blue-pc
#
#5. swap row0-shuffler-gold-p3 --- row1-shuffler-gold-p1
#6. swap row0-shuffler-gold-p7 --- row1-shuffler-gold-p5
#7. swap row0-shuffler-gold-pb --- row1-shuffler-gold-p9
#8. swap row0-shuffler-gold-pf --- row1-shuffler-gold-pd
```

For a four-row deployment, the only cable connections that must be changed upon the addition of a new suite are:

```
#0. swap row0-shuffler-blue-p0 --- no change
#0. swap row0-shuffler-blue-p2 --- row1-shuffler-blue-p0
#0. swap row0-shuffler-blue-p4 --- row2-shuffler-blue-p0
#0. swap row0-shuffler-blue-p6 --- row3-shuffler-blue-p0
#0. swap row0-shuffler-blue-p8 --- no change
#0. swap row0-shuffler-blue-pa --- row1-shuffler-blue-p8
#0. swap row0-shuffler-blue-pc --- row2-shuffler-blue-p8
#0. swap row0-shuffler-blue-pe --- row3-shuffler-blue-p8

#1. swap row1-shuffler-blue-p0 --- row0-shuffler-blue-p2
#1. swap row1-shuffler-blue-p2 --- no change
#1. swap row1-shuffler-blue-p4 --- row2-shuffler-blue-p2
#1. swap row1-shuffler-blue-p6 --- row3-shuffler-blue-p2
#1. swap row1-shuffler-blue-p8 --- row0-shuffler-blue-pa
```

```
#1. swap row1-shuffler-blue-pa --- no change
#1. swap row1-shuffler-blue-pc --- row2-shuffler-blue-pa
#1. swap row1-shuffler-blue-pe --- row3-shuffler-blue-pa

#2. swap row2-shuffler-blue-p0 --- row0-shuffler-blue-p4
#2. swap row2-shuffler-blue-p2 --- row1-shuffler-blue-p4
#2. swap row2-shuffler-blue-p4 --- no change
#2. swap row2-shuffler-blue-p6 --- row3-shuffler-blue-p4
#2. swap row2-shuffler-blue-p8 --- row0-shuffler-blue- pc
#2. swap row2-shuffler-blue-pa --- row1-shuffler-blue-pc
#2. swap row2-shuffler-blue-pc --- no change
#2. swap row2-shuffler-blue-pe --- row3-shuffler-blue-pc

#3. swap row3-shuffler-blue-p0 --- row0-shuffler-blue-p6
#3. swap row3-shuffler-blue-p2 --- row1-shuffler-blue-p6
#3. swap row3-shuffler-blue-p4 --- row2-shuffler-blue-p6
#3. swap row3-shuffler-blue-p6 --- no change
#3. swap row3-shuffler-blue-p8 --- row0-shuffler-blue-pe
#3. swap row3-shuffler-blue-pa --- row1-shuffler-blue-pe
#3. swap row3-shuffler-blue-pc --- row2-shuffler-blue-pe
#3. swap row3-shuffler-blue-pe --- no change
```

Following the completion of the above-described actions, sample connection information from the fiber-shuffler connection DB may look like:

```
LEAF: lr0c0a
lr0c0a.0a sr0c3a.0a
lr0c0a.0b sr0c3b.0a
lr0c0a.0c sr0c3a.4a
lr0c0a.0d sr0c3b.4a
LEAF: lr0c5b
lr0c5b.0a sr0c3a.2d
lr0c5b.0b sr0c3b.2d
lr0c5b.0c sr0c3a.6d
lr0c5b.0d sr0c3b.6d
```

In the above sample connection information, the entry `lr0c0a.0a sr0c3a.0a` may be interpreted as indicating that `leaf-row0-col0-leafInstance:a.port0-breakoutPort:a` connects through a fiber-shuffler to the spine interface `spine-row0-col0-port3-spineInstance:a.port0-breakoutPort:a`.

Thus, based on Link Layer Discovery Protocol (LLDP) information on a leaf and spine, and the local node (spine or leaf) information, there will be one, and only one, entry in the fiber-shuffler connection database that matches the lookup. The key for a lookup is a three-tuple, comprising the LLDP endpoints and the local node. For example, in the key "`0a, 2d, lr0c0a`" the value "`0a, 2d`" is extracted from the LLDP neighbors and the value "`lr0c0a`" is the local node relative to the LLDP neighbor. Such a lookup in the fiber-shuffler DB that was depicted above would result in "`sr0c3a`."

The result "`sr0c3a`" then provides the switch's information (a spine in this case), the switch's location (i.e., `r0c3`), which turns out to be the rack location as well, and the node instance (instance `a` in this case).

The next portion of the instant narrative provides a detailed flow for a rack discovery process.

As illustrated in Figure 1, above, the racks are interconnected using multiple fiber-shufflers. In order to achieve full mesh connectivity, the deployment is divided into blue and gold backbones. As shown in the figure, each rack is connected to both blue and gold backbones thru their respective fiber-shufflers. During rack discovery, the techniques presented herein leverage the fact that, when a leaf switch or a spine switch is discovered, based on the physical port on which that switch is discovered its physical location would be known and hence the rack (leaves and spines are physically hosted inside of a rack) may be discovered.

Adding a new rack in the same row (i.e., suite) is straightforward. The leaves of a rack in the same suite must be connected to the fibers-shufflers as shown in Table 1, below (which, once again, uses the arrangement that was depicted in Figure 1, above).

Table 1: Illustrative Connections

Fiber-Shuffler	Leaf
`r0a` `r0b`	Rack 0, Leaf A Rack 0, Leaf B
`r1a` `r1b`	Rack 1, Leaf A Rack 1, Leaf B
. .	. .
`r7a` `r7b`	Rack 7, Leaf A Rack 7, Leaf B

With the understanding that racks or new suites may be added on the fly, the addition of a new row or suite starts with the addition of "Spine Racks" and then the leaf racks are expanded on the either side of spine racks. The addition of new suites and racks must still follow the full mesh topology, where all of the leaves in the entire deployment are connected to all of the spines.

The cable swap logic that is needed here is the same that was used for the static DB generation (as described above). This is because the `row0-shuffler` would relate to all "`p0`'s" (in an 8x8 deployment), the `row1-shuffler` would relate to all "`p1`'s" (in an 8x8 deployment), and so on.

As noted previously, aspects of the techniques presented herein support the detection of mis-cabling between a leaf and spines and a fiber-shuffler.

As depicted in Figure 1, above, the leaves and spines in a rack must be connected to the slots as marked on a fiber-shuffler. For example, an `r0a` label on a fiber-shuffler implies that `row 0, leaf a, r6b - row 6, leaf b, p0, p2, p4, p6, p8, pa, pc, pe` are connected to "`spine a`" and `p1, p3, p5, p7, p9, pb, pd, pe` are connected to "`spine b`." Despite clear labeling on the fiber-shuffler, there is a possibility of human error. In the case of such a situation, it is desirable to detect the error as soon as possible.

Aspects of the techniques presented herein support an algorithm that may be employed to detect and handle a mis-cabling issue. Such an algorithm encompasses two assumptions.

First, the two leaves of one rack stay together due to a cabling constraint.

Second, a ShufflerRackDB is updated at every rack creation or deletion event. This DB is the source of truth for all of the racks that are discovered within a deployment and includes every bit of information that is related to a rack. Before admitting a rack to the discovered list, it may be run through a mis-cabling detection algorithm (as will be described below).

The following pseudo code presents one way in which the above-described algorithm may be expressed:

```
detectMiscabling(discoveredNode) {
// leaf or spine is discovered

// Serial number is the unique identifier for a leaf or spine
1. lookup(ShufflerRackDB) for serialNumber -> serialKeyEntry
2. lookup(ShufflerRackDB) for rackName -> rackKeyEntry
3. lookup(ShufflerRackDB) for nodeName -> nodeNameLookup
if serialNumber exists {
  if rackName exists {
    // RackName, SerialNumber and also the node name is same
    if discoveredNode == serialKeyEntry.NodeInfo.NodeName {
      // same leaf/spine discovered again, nothing to do
    } else {
      // nodea and nodeb in the same rack got interchanged
      // mis-cabling of 2 leaves with in the same rack
      // Example ...
      // 1.nodea with seriala in rack r0c3
      // 2.nodeb with serialb in rack r0c3
      // 1 and 2 steps are already discovered
      // 3. nodea is discovered as serialb in the same rack r0c3

      // Delete both the entries and add the new entry
      // in the "ShufflerRackDB"
    // (Expectation is we will get an update for the other leaf too)
    }
  } else {
    // This condition means same leaf/spine which was discovered in
    // rack "x" has been re discovered as rack "y" now.
    // UID remains the same, just the name gets updated. UUID is the
    // unique identifier for the rack, based on which
```

```
          // the system has been programmed. Name is more of a logical entity,
          // used for logging/data purposes only.
          // At the point, the expectation is that "we expect the other leaf will
          // also show up with the same updated rack name", eventually
        }
      } else {
        if rackName exists {
          if nodeNameLookup {
            // Leaf has been replaced in the existing rack.
            // We discover the same leaf with same name, but a different
            // serial number.
            // Delete the old entry.
          }
        // Update the new entry with the serial number.
      } else {
        // Node(leaf/spine) discovered for the first time
        // Generate a UUID for new rack
        }
        // Finally for all the cases, the updated entry is recorded into the DB.
        // add new entry to the ShufflerRackDB
        }
        return nil
      }
```

For purposes of exposition, several sample entries from an exemplary ShufflerRackDB (based on the dynamic rack discovery) are shown below:

```
[
{NodeInfo:{NodeName:sr1c3a Instance:a SerialNumber:spineSerial2 NodeType:spine}
RackInfo:{RackUUID:f0454fd1-a448-4ef7-93f2-10cc6fefd314 RackName:r1c3}}
{NodeInfo:{NodeName:sr0c3a Instance:a SerialNumber:spineSerial1 NodeType:spine}
RackInfo:{RackUUID:d481674d-d72d-406a-b33d-739b9d1150ac RackName:r0c3}}
{NodeInfo:{NodeName:lr0c3a Instance:a SerialNumber:leafSerial4 NodeType:leaf}
RackInfo:{RackUUID:d481674d-d72d-406a-b33d-739b9d1150ac RackName:r0c3}}
{NodeInfo:{NodeName:lr0c3b Instance:b SerialNumber:leafSerial2 NodeType:leaf}
RackInfo:{RackUUID:d481674d-d72d-406a-b33d-739b9d1150ac RackName:r0c3}}
]
```

According to the techniques presented herein, a simple tool that may be employed to detect mis-cabling may encompass a number of requirements. First, in an application centric infrastructure the leaves and spines operate within a management Internet Protocol (IP) setup and all of the ports on an individual switch (i.e., there are no partial connections within a rack) are fully connected to the shuffler ports (as shown in Figure 1, above). Second, the tool is not even aware of any shuffler device that may be used. Third, the shuffler connections DB (i.e., a static DB) contains all of the possible connection information across all of the kinds of deployments and such a DB may be embedded into the tool. Fourth, an input to the tool may be a configuration file, several sample entries from which are presented below:

```
---
- name: sr0c3a
type: spine
IP: 10.1.1.1
- name: lr0c3a
type: leaf
IP: 10.1.1.2
- name: lr0c3b
type: leaf
IP: 10.1.1.3
```

Based on the above-described requirements, the tool may, for every node that is defined in the configuration file, perform a series of steps. Under a first step, the tool may obtain the LLDP neighbors' information on the node, with the help from the (above-described) management setup. During a second step, a local switch DB may be formed, containing the neighbors' information, for every interface and its sub-interface. During a third step, the local switch DB (which was described in connection with Step 2) may be compared against the shuffler connections DB (i.e., the source of truth) that was described above. Under a fourth step, if a mismatch is detected (during Step 3) then it may be flagged. Finally, at a fifth step if the local switch DB matches the shuffler connections DB, then the focus may be moved to the next switch.

As described and illustrated above, the innovative use of known elements (including fabric shufflers and the discovery of racks) enable the techniques presented herein to support, among other things, the automatic discovery of a newly added rack within a data center. Specifically, the way in which the addition of a rack may be detected (through use of aspects of the presented techniques) is not currently supported within an application centric infrastructure and no solution is known that uses the presented method. Additionally, while any re-cabling or cable swapping may happen manually (or by the machines) on the addition of a new rack, aspects of the presented techniques support the automatic detection of mis-cabling on a newly added rack.

In summary, techniques have been presented herein that aid in the automatic detection of mis-cabling issues, the automatic discovery of a rack's location within a large-scale data center, and the automatic detection of the addition of a new rack to a data center, all without disrupting network traffic within a dynamically expanding fabric. Aspects of the presented techniques provide an algorithm that can automatically detect such mis-cabling issues. Further aspects of the presented techniques support a tool that may be used to detect mis-cabling (e.g., in the case of inter-communication failures among the nodes).