Universidad de Alcalá
BIBLIOTECA

Document downloaded from the institutional repository of the University of Alcala: http://ebuah.uah.es/dspace/

This is a postprint version of the following published document:

Rojas, E., Álvarez Horcajo, J., Martínez Yelmo, I., Arco, J.M. & Carral, J.A. 2017, "GA3: scalable, distributed address assignment for dynamic data center networks", Annals of Telecommunications, vol. 72, pp. 693-702.

Available at https://doi.org/10.1007/s12243-017-0569-4

© 2017 Springer Nature

(Article begins on next page)

This work is licensed under a

Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.

Annals of Telecommunications manuscript No.
(will be inserted by the editor)

GA3: Scalable, distributed address assignment for dynamic data center networks

Elisa Rojas · Joaquin Alvarez-Horcajo · Isaias Martinez-Yelmo · Jose M. Arco · Juan A. Carral

Received: date / Accepted: date

Abstract Deployment and maintenance of current data center networks is costly and prone to errors. In order to avoid manual configuration, many of them require centralized administrators which constitute a clear bottleneck, while distributed approaches do not guarantee sufficient flexibility or robustness. This paper describes and evaluates GA3 (Generalized Automatic Address Assignment), a discovery protocol that assigns multiple unique labels to all the switches in a hierarchical network, without any modification of hosts or the standard Ethernet frames. Labeling is distributed and uses probes. These labels can be leveraged for shortest path routing without tables, as in the case of the Torii protocol, but GA3 also allows other label-based routing protocols (such as PortLand or ALIAS). Additionally, GA3 can detect miswirings in the network. Furthermore, control traffic is only necessary upon network deployment rather than periodically. Simulation results showed a reduced convergence time of less than 2 seconds and 100kB/s of bandwidth (to send the GA3 frames) in the worst case for popular data center topologies, which outperforms other similar protocols.

Keywords Data centers · Automatic address assignment · Misconfiguration detection · Shortest Path Bridges

E. Rojas
E-mail: elisa.rojas@telcaria.com

J. Alvarez-Horcajo
E-mail: j.alvarez@uah.es

I. Martinez-Yelmo
E-mail: isaias.martinezy@uah.es

J.M. Arco
E-mail: josem.arco@uah.es

J.A. Carral
E-mail: juanantonio.carral@uah.es

1 Introduction

Data Center Networks (DCNs), which constitute the main pillar of most major Internet services, are increasingly relying on Ethernet and flat layer-two networks due to the excellent price/performance ratio and ease of configuration. However, legacy Ethernet layer-two networks do not scale. To solve this, the most common approach is to assign globally unique, topologically meaningful host *labels* [1] that the network applies internally for forwarding. Labeling simultaneously yields the key benefits of IP addressing (hierarchical address assignments for simpler forwarding tables) and of MAC addressing (no need for manual configuration).

Following this criterion, a network designer must consider a vast number of topology families and architectures [2], which usually have their own labeling and routing protocols. These solutions are often very specific and relatively inflexible because they are designed with a fixed topology. However, a network administrator might need to expand or reorganize the topology or cope with network failures, which can happen frequently as the amount of network equipment increases [3] and cause service interruptions that may have severe consequences. Furthermore, topological flexibility can improve traffic optimization and simplify network management [4], and even non-regular topologies might show high performance benefits [5].

Therefore, a key concern in DCNs is automatic configuration in the face of a dynamically changing topology [6]. Unfortunately, distributed proposals lack this desirable feature and SDN-based solutions still present scaling limitations [7]. Currently, only ALIAS [1,6] satisfies these requirements, but it has several limitations: an appreciable amount of traffic is required maintain la-

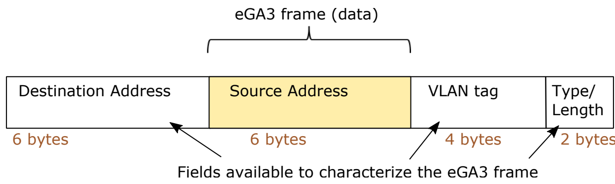

Fig. 1 GA3 frame format

beling and it lacks the capacity to detect possible topology miswirings.

In this paper, we present GA3, a generalized labeling protocol for DCNs. GA3 was originally designed for the Torii routing protocol [8], but it has been extended to cover a wider range of DCN topologies. GA3 is a new labeling paradigm that explores and labels the network rapidly and flexibly, thus reducing message load since messages are only exchanged in the set up rather than periodically.

The structure of this paper is as follows: In Section 2, we describe GA3, and assess it in Section 3. In Section 4, we discuss related work, and in Section 5 we summarize and conclude the paper.

2 GA3

GA3 is a fully distributed protocol for address assignment, labeling switches in hierarchical networks, including DCNs. It uses *probe* frames that explore and inform all the nodes of a topology, following the All-Path locking mechanism [9] to avoid loops. GA3 frames (Fig. 1) are common Ethernet frames that reuse the source MAC address field to disseminate this information. GA3 frames can be distinguished by: either a specific multicast address as destination MAC address, a VLAN tag or specific Ethertype. The frame contains 6 bytes in data, where we save the first 4 bits to differentiate up to 16 types of frame (including Hello or SetHLMAC, which assigns the labels), leaving 44 bits available for use within GA3.

2.1 Address assignment

GA3 assigns Hierarchical Local MAC (HLMAC) addresses (as defined in Torii [8]) carried within *probe* frames that explore the entire network, starting from the core switches.

Before explaining the assignment method, let us explain the terminology:

- *HLMAC addresses* are composed of three different elements:
  - *Prefix*: The first field of the HLMAC, which identifies the core or root (`1.` in the case of `1.2.3.4.`)
  - *Suffix*: The following fields, excluding the last one, which identify the branch or path toward a switch in the network (`2.3.` in the case of `1.2.3.4.`)
  - *Coordinate*: The last field, which identifies the device (`4.` in the case of `1.2.3.4.`)
- HLMAC addresses have a *priority level*, which indicates what address has preference and helps decide if a GA3 frame should be flooded.
  Addresses are compared field by field (excluding the prefix) and the first one that is different determines the priority. The lower the value, the higher the priority. For example, `1.2.3.` has higher priority than `1.2.4.`, `1.2.3.4.` (omitted values after the dot are zeroes), or even `1.3.`; however, it has lower priority than `1.2.2.`, `1.1.2.3.`, or `1.1.`
- HLMAC addresses have a *hierarchy level*, which is the number of values that the address contains (minus the prefix). For example, `1.` has a hierarchy level 0 (core switch), while `1.2.3.4.` has a hierarchy level 3. Hosts obtain HLMAC addresses with the highest hierarchy level in the topology.
  In particular, GA3 considers one byte per hierarchy level; for instance, `1.2.3.` uses 3 bytes: one for 1, another for 2, and another one for 3. This is the most common and standard behavior, as stated in [10], which is why it is used in GA3.

Below, we present increasingly complex examples to illustrate how GA3 creates the HLMAC addresses, transmits them, and assigns them throughout the network.

2.1.1 Basic address assignment example

The process starts at the core switch with the highest HLMAC priority. It broadcasts a SetHLMAC containing its own HLMAC address plus a coordinate. This coordinate can be the port number from which it is being sent or an equivalent, but it must be unique per switch. The process is repeated for all the switches that receive the SetHLMAC message. For example, in Fig. 2.a, S1 is the core switch and contains the HLMAC `1.`, so it sends a SetHLMAC containing HLMAC `1.9.` because it is sent from port 9. S3 receives it and associates that address with the incoming port. After this, it broadcasts the frame through ports 7 and 6 (adding those coordinates respectively), and switches S5 and S6 assign addresses `1.9.7.` and `1.9.6.` to their input ports, respectively. Note the assigned HLMAC is the chain of designated port IDs traversed in the descending path from the core switch to the device.

Synchronization of suffixes from multiple cores

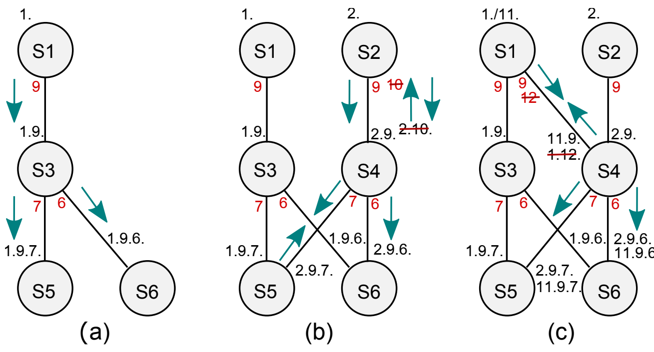

Fig. 2 Basic address assignment example

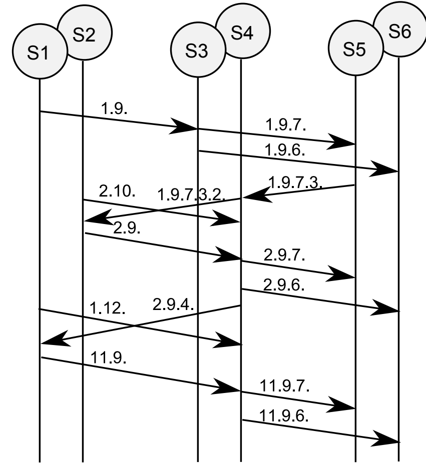

Fig. 3 Sequence diagram of the basic address assignment example

A new core with new branches has been added symmetrically in Fig. 2.b. The initial presumption might be that the process is exactly the same, once per core. However, this would cause switches to be given HLMAC addresses that cannot be inferred from the others just by changing the prefix (required by some routing protocols), because the suffix depends on the IDs of the ports traversed, and these might have different values. For example, S5 could obtain `1.9.7.` and `2.10.5.`, which have different suffixes.

To synchronize the suffixes, GA3 learns labels and floods only if the HLMAC priority from the SetHLMAC frame is the highest received so far. Therefore, S5 and S6 continue flooding the network, sending SetHLMAC messages with values `1.9.7.5.` and `1.9.6.6.`, respectively, to S4. Then, S4 sends children of these HLMAC addresses to S2, which receives addresses starting from `1.9`. S4 still does not know if it is a child of S5 or S6, or if its HLMAC `2.10.` is correct. However, S2 does know that it is a core switch and it will not assign the HLMAC addresses received from S4; but it will use them to synchronize port numbering: since S2 is `2.`, it knows that the following hierarchy levels from that branch should start with 9 (eliminating the 1 from the received `1.9.`).

When S2 propagates its children HLMAC addresses, it knows that the HLMAC to be sent should be `2.9.` (instead of the original `2.10.`) thanks to the SetHLMAC copy that arrived from S1. S4 then associates `2.9.` with its port of origin and sends `2.9.7.` and `2.9.6.` to S5 and S6, respectively. S4 knows it should use the values 7 and 6, because it previously received `1.9.7.5` and `1.9.6.6.`, and since its hierarchy level is 2, it can remove the `1.9.` and obtain the corresponding values.

In the case that S2 starts sending the SetHLMAC before receiving the synchronization information, it would send `2.10.` and the process would continue until reaching the edge switches. However, whenever S4 received `1.9.7.5.` or `1.9.6.6.`, it would acknowledge them as having a higher priority level than any other starting from `2.` and would therefore continue flooding them until reaching S2, which would then restart the process using `2.9.` instead of `2.10`.

Lastly, S5 and S6 have HLMAC addresses `1.9.7.`, `2.9.7.` and `1.9.6.`, `2.9.6.`, which can be deduced from the others simply by changing the prefix (i.e. suffixes and coordinates are the same per switch).

Duplicated paths from a common core

We illustrate the last part of the process in Fig. 2.c. In this example, we have added another link between switches S1 and S4. Now, switches S3, S4, S5, and S6 have two paths to reach S1. At first, S1 sends a SetHLMAC with the value `1.9.` through one port and `1.12.` through the other, because it does not know if both ports are linked to a common pod or not. However, this behavior would once again create HLMAC addresses with different suffixes (not inferable); therefore, S1 should send SetHLMAC messages that only vary in their prefix to links with commons pods. To do this, S1 creates virtual prefixes (or *virtual cores*), e.g. `1.` has `11.`, `21.`, etc.

The typical scenario of a virtual core is the following: S1 sends SetHLMAC messages through both ports and eventually these will reach each other at some switch in the network. Since any address starting from `1.9.` has a higher priority than `1.12.`, the SetHLMAC containing `1.9.` and children will continue its propagation while the ones from `1.12.` will be discarded. Eventually, a copy arrives at S1 again in some ports (in this case, in the link connected to S4) and this notifies S1 that both the link from S1-S3 and S1-S4 share a common pod (otherwise no copy would have arrived), and therefore S1 again sends a new SetHLMAC to S4 but this time containing `11.9.` instead of `1.12`.

Finally, S5 obtains `1.9.7.`, `11.9.7.` and `2.9.7.`, and S6 obtains `1.9.6.`, `11.9.6.` and `2.9.6.`

Fig. 3 illustrates a summary of this example in the form of a sequence diagram, which helps us understand the temporal exchange of messages from Fig. 2. The first SetHLMAC goes from S1 to S3 and contains `1.9.`, while the last messages go from S4 to S5 and S6.

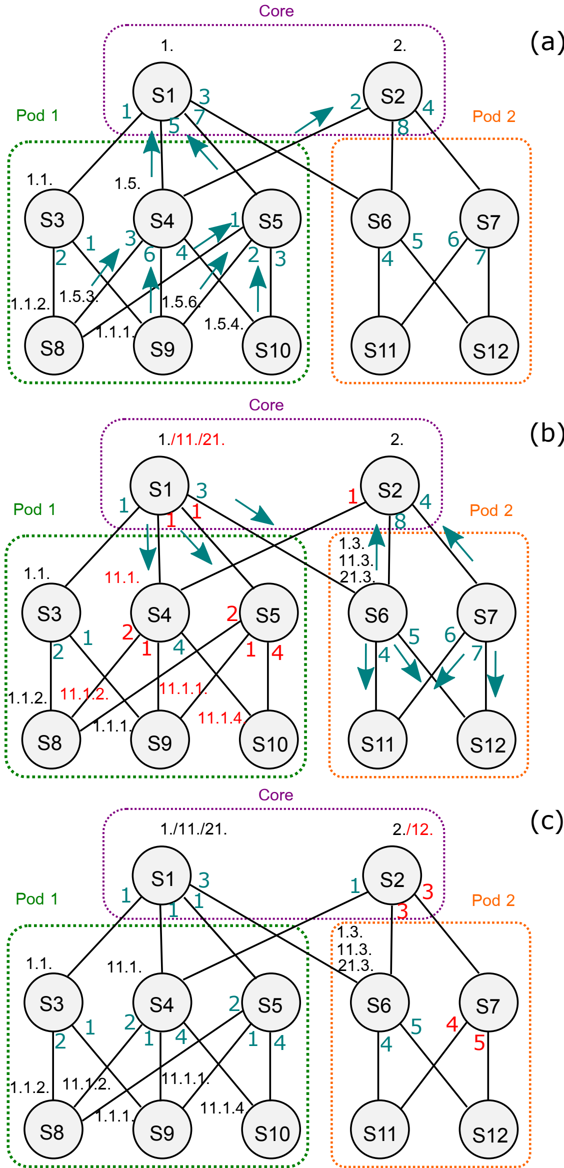

Fig. 4 Extended address assignment example

2.1.2 Extended address assignment example

Let us nos consider Fig. 4: a topology with two cores, two pods[1] and five hosts. S1 (core 1) starts sending two SetHLMAC messages through ports 1 and 5 and assigns `1.1.` and `1.5.` to S3 and S4, respectively. Once received, S3 and S4 send SetHLMAC messages to S8, S9, and S10. S10 only obtains one HLMAC, `1.5.4.`, but S8 and S9 obtain two: one from S3 and one from S4, assigned to two different ports. For example, S8 associates `1.1.2.` with one port and `1.5.3.` with the other and because the former has a higher priority, it will generate and send a SetHLMAC to S4, while the latter will not (see Fig. 4.a)[2]. A similar logic is applied at S9, which forwards a child SetHLMAC of `1.1.1.` to S4, but no child of `1.5.6.` to S3 as its priority is lower. Therefore, children of `1.1.` arrive at S4 and eventually return to S1. The same happens with children of `1.1.` arriving at S5.

The next step is that S1 creates two new virtual core prefixes and sends `11.1.` to S4 and `21.1.` to S5. These SetHLMAC messages propagate until reaching the edge switches, which receive new HLMAC addresses with suffix synchronization (updating port IDs). Fig. 4.b shows the resulting addresses with changes marked in red.

In the meantime, a copy of the SetHLMAC also arrives at the second pod, at S3 as `1.3.`, and different children flood the pod until reaching S2 (core 2), which also updates different port IDs in S2 and S7 (see Fig. 4.c). Apart from `1.3.`, two other SetHLMAC with addresses `11.3.` and `21.3.` will also arrive at S3 after S1 generates the new virtual core prefixes.

The final address assignment at edge level for hosts is shown in Fig. 4.c. Note that all hosts have up to five addresses except for H3, which has four because there is no connection between switches S3 and S10.

Multiple virtual core prefixes

A core creates a new virtual prefix per path to a common switch. Depending on the duplicated paths from a core to a switch, the number of virtual core prefixes can grow exponentially; however, this number can easily be reduced by not propagating all HLMAC addresses, which implies not generating all different paths available. By default, GA3 explores all possible paths between pairs of devices, so decreasing the number of HLMAC addresses propagated reduces messaging, which is particularly desirable in networks containing a high number of cores.

For example, if we add a link between S1 and S7 in Fig. 4.c, S1 will have multiple paths in the first pod and in the second. In this case, S1 sends HLMAC addresses with prefixes `1.`, `11.`, and `21.` to both S6 and

[1] A pod is a group of switches that share connectivity via links not coming from a core. S4 and S5 belong to the same pod; but not S5 and S6, as they are only connected through the core S1.

[2] It might happen that `1.5.4.` arrives earlier than `1.1.2.` and is then propagated to S3, but it would subsequently be stopped at S3, which has a higher priority: `1.1.`

S7. Eventually, the loop is discovered and one set of addresses will have a higher priority than the other, so S1 generates virtual cores. How many? Ideally, one virtual prefix per original prefix, which makes a total of six. However, the core might also decide not to create so many new prefixes (by setting a parameter `B`). The network administrator should define `B` looking for a tradeoff between the number of available paths required and the amount of messages exchanged.

2.1.3 Pseudocode

We summarize the study of GA3 in a pseudocode, which explains the actions that a switch takes after receiving a SetHLMAC message. A **switch** contains a list of `core` prefixes, a list of `ports` and a priority (`prio`), which is the highest of the `prio` values in the port list. Each switch **port** contains a physical (`phy`) value (manufacturer MAC), a logical (`id`) value, a list of assigned `HLMAC` addresses and a priority (`prio`), which is the assigned HLMAC with the highest priority, without the prefix. The hierarchy level (`hier`) is the length of the prio value (number of bytes).

```
1  receive_message(SetHLMAC, input_port)
2  {
3   port = switch.ports[input_port]
4
5   if (port.hier >= SetHLMAC.hier):
6
7   #port: update HLMAC list
8   if port.prio.suffix() != SetHLMAC.suffix():
9     port.HLMAC.clear()
10  port.HLMAC.append(SetHLMAC.hlmac)
11
12  #port: update prio
13  if port.prio < SetHLMAC.prio:
14    port.prio = SetHLMAC.prio
15
16  #switch: update prio
17  if switch.prio < SetHLMAC.prio:
18    switch.prio = SetHLMAC.prio
19          flood(SetHLMAC, input_port) #forward
20
21  #ports: update ids
22  ports = []
23  for p in switch.port:
24    if p.hier > switch.hier &&
25    (p.hier-switch.hier<=2 || switch.core):
26          p.id = p.prio[switch.hier]
27    if p.id in ports && switch.core != null:
28          switch.core.append(new_core)
29          startcore(new_core, p) #restart core
30    else
31          ports.append(p.id)
32 }
33
34 flood(msg, input) #forward SetHLMAC
35 {
36  for port in switch.ports:
37    if port != input:
38      send (msg + port.id, port.phy)
39 }
40
41 startcore(core, port)   #startup or virtual
42 {
43  send (core + port.id, port.phy)
44 }
```

2.2 Protocol features

Having described GA3 address assignment, in this section we will now focus on the different properties of the protocol.

2.2.1 Label robustness

GA3 assigns addresses upon deployment of the network, but it does not need to repeat the process. Changes in the network only invoke local changes in the addressing, yielding high adaptability, fast recovery, and very low bandwidth required for the exchange of GA3 messages.

If a link goes down or is removed from the network, only the adjacent nodes need to delete the addresses associated with the affected ports. Similarly, in the case of switches, these affect a set of links and again only the adjacent nodes delete the assigned addresses. If a link or a switch goes up or is added to the network, neighbors will discover them (via *Hello* messages) and send a SetHLMAC message only through the recently activated ports. This message propagates locally until finding an already assigned, higher priority HLMAC, allocating new HLMAC addresses along its route.

Thus, no global reformulation of addresses is required, increasing system scalability and reducing maintenance.

2.2.2 Miswiring triggers

The G3A procedure is not only capable of assigning ordered HLMAC addresses, but can also recognize topology patterns and raise alerts when this is not the desired topology, that is, it can recognize different types of possible miswirings or faulty ports:

- **Loops in a pod:**
  If a core switch receives an upstream SetHLMAC that contains a HLMAC address assignment with its same core prefix, this indicates that this core switch has two or more links to a common pod, one of the pods is not completely isolated, or there are no pods at all.
  As already mentioned, this generates what is called a *virtual core* and does not represent a miswiring per se. However, if the network administrator was setting up a fat-tree [11] topology, which has well-defined pods with just a single link to core switches, then this would indeed indicate a topology deployment error.
- **Irregular core assignment:**
  If an edge switch does not have all the possible HLMAC addresses after address assignment, the topology is not symmetric or the network is not optimal.

- **Peer links:**
  If a switch has a pair of addresses that only differs in one hierarchical level after an update, this indicates that *peer links* exist in the network, i.e. links that directly connect switches in the same hierarchy level (for example, a link between two core switches), which could be considered misconfiguration for data center topologies.

Switches could report this information individually at switch level or to a centralized manager, so the network administrator would be able to check the deployment and determine whether the setup is correct.

2.3 Trade-off between convergence time and exchanged data

The actual number of messages in a deployment depends on an extra parameter: the *GA3-clock*. A GA3-clock equal to zero means that SetHLMAC messages are all flooded at the same time, while any other value represents an extra delay when forwarding the frame in different ports, using the ports with lower ID first. For example, if a switch floods the frame through ports 1, 2 and 3, GA3-clock=0 sends all the frames at the same time and GA3-clock=X sends through port 1 at time 0, port 2 at time X, and port 3 at time 2*X. Additionally, this delay can be weighted based on the hierarchical level of the switch (increasing it at core and decreasing it at edge level).

Any value higher than zero for the GA3-clock reduces the total number of messages in the network: the higher the value, the lower the number of messages exchanged. Suffix synchronization is performed prior to sending other messages and thus, GA3 avoids resending extra messages.

The GA3-clock is especially important in this field because discovery and label assignment are usually carried out by protocols with periodical retransmission intervals. These protocols (such as LLDP [12]) exchange data messages between neighbors and their performance depends on their periodicity. However, GA3 can either depend on a clock or be completely independent from it: if the GA3-clock equals zero, the convergence time will only depend on the number of hops in the network and the transmission rate of the links. Thus, network administrators can always configure the GA3-clock as a trade-off.

3 Evaluation

In this section report an analytical evaluation and a simulation-based evaluation of GA3, implemented in Python. In both cases, we compared the number of messages generated by GA3 and ALIAS, which we believe to be the closest proposal to date to our own work. In addition, we took PortLand [13] and VL2 [14] (with $Dc = Da = k$) as examples of common data center topologies. The reason for this is that the fat-tree from PortLand inspired some of the data center deployments in Google [15], while VL2 is one of the most famous designs from Microsoft [16]. Many other surveys, such as [17,18], consider both topologies as representative of data center network architectures.

3.1 Analytical evaluation

We analyzed GA3 in the range between "near best" and "near worst" cases, given that the number of messages generated by GA3 depends on the GA3-clock. Note that GA3 only generates messages when the network is initialized, whereas ALIAS produces them periodically (i.e. the process is repeated indefinitely depending on the period parameter defined in ALIAS, whereas GA3 only needs one execution). Finally, let us define C as core, A as aggregation, and E as edge switches.

GA3 in PortLand

Near best: In this case, we assumed that upward messages with the highest priority arrive first at A and it forwards the messages to C, then the lower priority messages arrive and are not forwarded. The first core sends one SetHLMAC. This message arrives at A and is forwarded down by the first A to the E through its $k/2$ ports (one message per port) to reach all the k/2 edges of the first pod. The total number of messages going down in that pod is: $DOWN = 1 + k/2$

Let us calculate the upward messages in that pod, which enable GA3 to synchronize the suffixes. Every edge node has $k/2$ links to A and sends messages by all ports except the one that received the downward message, so $k/2 - 1$. Multiplied by $k/2$ edge nodes per pod: $UP_E = k/2 * (k/2 - 1) = k^2/4 - k/2$. Apart from E, A also sends $k/2$ (one per port) messages upward, except the node that receives the frame from the core, which sends $k/2 - 1$, i.e.: $UP_A = k/2 - 1 + (k - 1) * (k/2 - 1)$. Therefore, the messages produced in a pod from one SetHLMAC: $TOTAL_{POD} = 3k^2/4 - k + 1$

If the first core repeats this process for its k ports and there are $k^2/4$ cores in the network, the total is:

$$E_{Pbest} = k * \frac{k^2}{4} * (\frac{3k^2}{4} - k + 1) = \frac{3k^5}{16} - \frac{k^4}{4} + \frac{k^3}{4} \quad (1)$$

Near worst: In this case, we assumed that upward messages are received the other way round: from the lowest to the highest priority, so that every message

Topology	GA3 "near worst"	GA3 "near best"	GA3 Hellos	ALIAS (periodical)
PortLand	$\frac{3k^5}{16} - \frac{k^4}{4} + \frac{k^3}{4}$	$\frac{k^6}{16} - \frac{k^5}{8} + \frac{k^4}{4}$	$4k^2$	$\frac{5k^3}{24}$
VL2	$B * (\frac{5k^3}{4} - \frac{k^2}{2})$	$B * (\frac{k^4}{4} - \frac{k^3}{2})$	$7k^2$	$14k^2$

Table 1 Comparison of number of messages (GA3 vs ALIAS)

needs to be re-forwarded. This behavior only affects the number of upward messages, because now A resends all the $k/2$ frames received from E, i.e.: $UP_A = k/2 - 1 + k/2*(k-1)*(k/2-1)$. Therefore, the messages produced in a pod from one SetHLMAC: $TOTAL_{POD} = k^3/4 - k^2/2 + k$

If the first core repeats this process for its k ports and there are $k^2/4$ cores in the network, the total is:

$$E_{Pworst} = k * \frac{k^2}{4} * (\frac{k^3}{4} - \frac{k^2}{2} + k) = \frac{k^6}{16} - \frac{k^5}{8} + \frac{k^4}{4} \quad (2)$$

GA3 in VL2

Near best: Similarly to PortLand, we assumed that the highest priority messages arrive first. Following the same reasoning, the number of messages produced from one SetHLMAC sent from one core is: $TOTAL_{POD} = 1 + k/2 + 2k - 2 = 5k/2 - 1$

If the first core repeats this process for its 2 loops in the pod, B loops by core, and there are $k/2$ pods and $k/2$ cores in the network, the total number of messages is:

$$E_{Vbest} = 2 * B * \frac{k}{2} * \frac{k}{2} * (\frac{5k}{2} - 1) = B * (\frac{5k^3}{4} - \frac{k^2}{2}) \quad (3)$$

Near worst: Similarly to PortLand, we assumed that upward messages are received the other way round, which only affects the number of upward messages. Following the same reasoning, the number of messages produced from one SetHLMAC sent from one core is: $TOTAL_{POD} = k^2/2 + k$

If the first core repeats this process for its 2 loops in the pod, B loops by core, and there are $k/2$ pods and $k/2$ cores in the network, the total number of messages is:

$$E_{Vworst} = 2 * B * \frac{k}{2} * \frac{k}{2} * (\frac{k^2}{2} + k) = B * (\frac{k^4}{4} - \frac{k^3}{2}) \quad (4)$$

ALIAS in PortLand

ALIAS sends two types of message periodically: pings (to check if the link is connected to a host or a switch) and TVM messages (to set the coordinates). The TVM process can create collisions and the number of messages would then increase (the procedure would be repeated for some of the labels), but we are considering the simplest case: counting only the initial TVM.

If every device (switches and hosts) in the network sends a ping and only switches send TVM to their neighbors, then the total number of messages is:

$$A_P = 5k^3/24 \quad (5)$$

ALIAS in VL2

If the case of ALIAS, the reasoning is the same, but the number of nodes in the network is different:

$$A_V = 14k^2 \quad (6)$$

Table 1 shows a comparison between GA3 and ALIAS of the number of messages needed for address assignment. Although the magnitude for ALIAS is k^2 and while for GA3 it is up to k^6, ALIAS is periodical and GA3 is executed just once; therefore, after k^5 repetitions, ALIAS would exceed GA3.

3.2 Simulation-based evaluation

The GA3 address assignment simulator was designed in Python (code available in [19]). The simulator gives two outputs: the assigned HLMACs on each node for the selected topology and the number of GA3 frames exchanged.

Topology	Simulation	Upper bound (Worst case)	Lower bound (Best case)
VL2 k=4	64	192	72
VL2 k=6	207	1296	252
Portland k=4	120	192	144
Portland k=6	1242	2268	1188
Irregular	117	—	—

Table 2 Summary of simulation results (Number of GA3 messages)

We selected the following three-level hierarchical topologies: VL2 with Da=Dc=k=4, VL2 with k=6 (Fig. 5), Portland with k=4 (Fig. 6), Portland with k=6, and an irregular hierarchical topology with a cross and a peer link (Fig. 7). GA3 successfully assigned HLMAC addresses in the topology in one round after set up and the number of messages was low.

The comparison is shown in Table 2. The simulation results presented even lower values than the "near

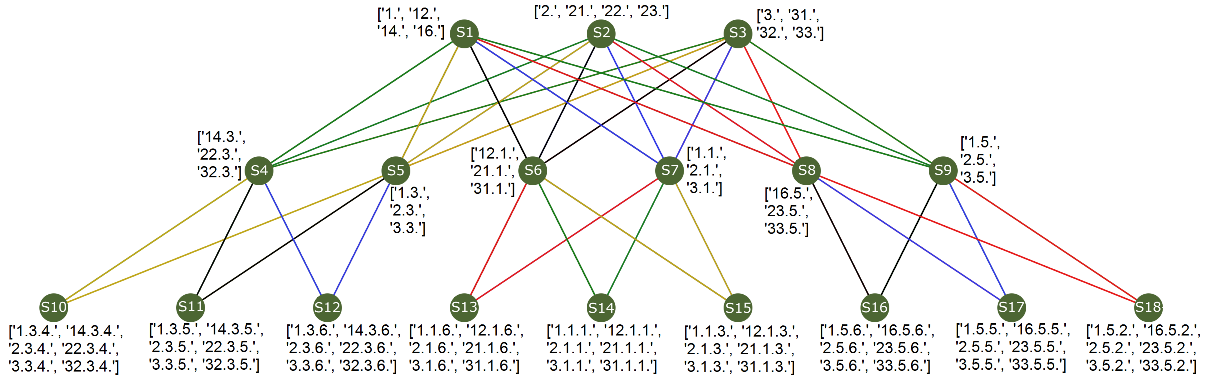

Fig. 5 VL2 with Da=Dc=k=6

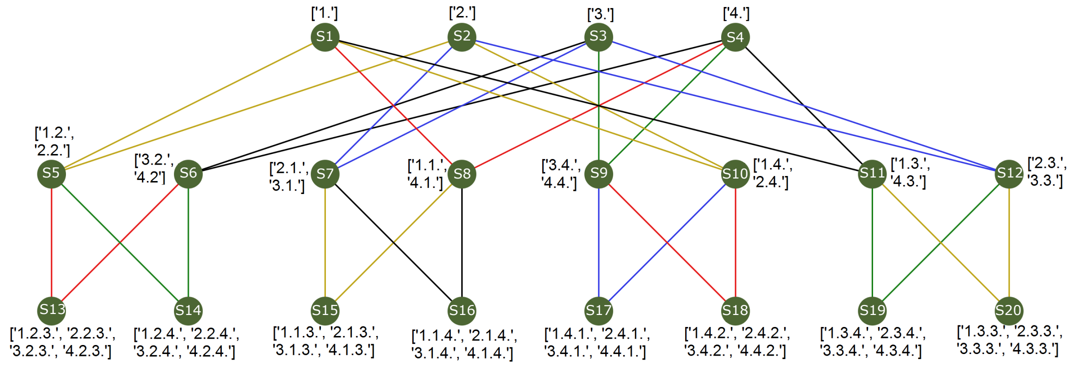

Fig. 6 Portland with k=4

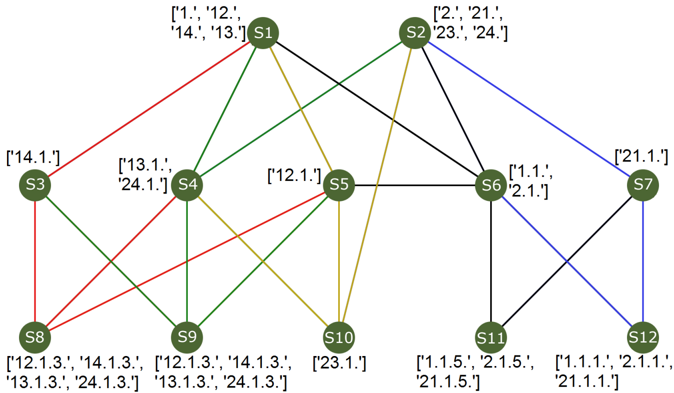

Fig. 7 Irregular hierarchical toplogy

best" cases because the simulator has the capacity to disregard some virtual cores after creating a number of multiple paths (explained in Section 3). This was not considered in the equations, but values are in the same range. If we consider a GA3-clock=1ms and a frame size of 64B, the total time required in the worst case would be <2 seconds and less than 100kB/s, which indicates a very low impact, especially considering the exchange is performed only once after network deployment.

4 Related Work

The most similar work in the literature to GA3 is the Decider/Chooser Protocol (DCP) from ALIAS [6], which automates topology discovery and address assignment for a wide range of hierarchical data center topologies. However, it exchanges messages periodically, is unable to detect miswirings and consumes more resources, as we proved in the evaluation section.

Regarding **topology discovery and address assignment**, the Link Layer Discovery Protocol (LLDP) [12] or other approaches based on SNMP [20] are neighbor discovery protocols that assist in discovering the physical topology. However, they do not have the capability to assign topological information or labels to network devices. Sourcey [7] discovers and monitors the network with server-only mechanisms, which requires modifications of final hosts and standard frames. Other proposals for data centers include: DAC [21], Tree-conf [22] and GARDEN [23]; but both DAC and Tree-conf require a blueprint of the topology to operate and GARDEN has a centralized controller bottleneck.

Related to **misconfiguration** detection, Batfish [24] can find errors proactively (before the configuration is applied) and answer "what if" questions, but it focuses on analyzing specific network configuration files rather

than topology cabling. DAC also detects miswirings, however it halts when malfunctions are detected, which then needs to be resolved manually. The Error Tolerant Address Configuration (ETAC) [25] is an optimized version of DAC, which generates a network graph by removing devices in the physical network involved in malfunctions, but these malfunctions are located manually and excluded devices are no longer used in the graph, regardless of their malfunction type, thus wasting resources. The Miswiring Tolerant Routing protocol (MTR) [26] surpasses the limitations of DAC and ETAC, but is only applicable to Clos-based networks.

5 Conclusion

GA3 is an efficient labeling protocol for hierarchical networks, including data centers, which assigns ordered HLMAC addresses with probe frames that discover the topology and explores all possible shortest paths in the network. GA3 is also capable of detecting miswirings and faulty ports. GA3 obtained very good evaluation results with a low control overhead, improving on the closest approach: DCP from ALIAS.

This solution opens a new field in topology discovery: it implements probe frames to explore the network instead of obtaining the information through periodical messaging. It also provides high adaptability for data center networks, because even if a link fails or the network is expanded, GA3 continues working in a distributed manner and, at the same time, it avoids manual configuration or centralized bottlenecks. Furthermore, our proposal is compatible with different architectures and forwarding approaches.

Acknowledgements This work was supported in part by grants from Comunidad de Madrid through Project TIGRE5-CM (S2013/ICE-2919)

References

1. M. Walraed-Sullivan, R. N. Mysore, M. Tewari, Y. Zhang, K. Marzullo, and A. Vahdat, "ALIAS: Scalable, decentralized label assignment for data centers," in *SoCC*. ACM, 2011, p. 6.
2. R. Agarwal, J. Mudigonda, P. Yalagandula, and J. C. Mogul, "An Algorithmic Approach to Datacenter Cabling," HP Laboratories, Tech. Rep. HPL-2015-8, February 2015.
3. R. Potharaju and N. Jain, "When the network crumbles: an empirical study of cloud network failures and their impact on services," in *SoCC*. ACM, 2013, pp. 15:1–15:17.
4. Y. Xia, M. Schlansker, T. S. E. Ng, and J. Tourrilhes, "Enabling Topological Flexibility for Data Centers Using OmniSwitch," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, Jul. 2015.
5. A. Curtis, T. Carpenter, M. Elsheikh, A. Lopez-Ortiz, and S. Keshav, "REWIRE: An optimization-based framework for unstructured data center network design," in *INFOCOM, 2012 Proceedings IEEE*, March 2012, pp. 1116–1124.
6. M. Walraed-Sullivan, R. N. Mysore, K. Marzullo, and A. Vahdat, "A Randomized Algorithm for Label Assignment in Dynamic Networks," Microsoft Research, Tech. Rep. CS2013-0994, February 2013.
7. X. Jin, N. Farrington, and J. Rexford, "Your Data Center Switch is Trying Too Hard," in *ACM Symposium on SDN Research (SOSR 2016)*, 2016.
8. E. Rojas, G. Ibanez, J. M. Gimenez-Guzman, D. Rivera, and A. Azcorra, "Torii: multipath distributed Ethernet fabric protocol for data centres with zero-loss path repair," *Transactions on Emerging Telecommunications Technologies*, vol. 26, no. 2, pp. 179–194, Feb. 2015.
9. E. Rojas, G. Ibanez, J. M. Gimenez-Guzman, J. A. Carral, A. Garcia-Martinez, I. Martinez-Yelmo, and J. M. Arco, "All-Path bridging: Path exploration protocols for data center and campus networks," *Computer Networks*, vol. 79, pp. 120 – 132, 2015.
10. R. M. Ramos, M. Martinello, and C. E. Rothenberg, "SlickFlow: Resilient source routing in Data Center Networks unlocked by OpenFlow," *38th Annual IEEE Conference on Local Computer Networks*, vol. 0, pp. 606–613, 2013.
11. M. Al-Fares, A. Loukissas, and A. Vahdat, "A scalable, commodity data center network architecture," *ACM SIGCOMM Computer Communication Review*, vol. 38, no. 4, pp. 63–74, 2008.
12. "IEEE 802.1AB (LLDP) Specification," 2009.
13. R. Niranjan Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat, "PortLand: A scalable fault-tolerant layer 2 data center network fabric," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 39–50, Aug. 2009.
14. A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, "VL2: A scalable and flexible data center network," *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 4, pp. 51–62, Aug. 2009.
15. A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat, "Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 4, pp. 183–197, Aug. 2015.
16. T. Chen, X. Gao, and G. Chen, "The features, hardware, and architectures of data center networks: A survey," *Journal of Parallel and Distributed Computing*, vol. 96, pp. 45 – 74, 2016.
17. R. Rojas-Cessa, Y. Kaymak, and Z. Dong, "Schemes for Fast Transmission of Flows in Data Center Networks," *IEEE Communications Surveys Tutorials*, vol. 17, no. 3, pp. 1391–1422, 2015.
18. W. Xia, P. Zhao, Y. Wen, and H. Xie, "A Survey on Data Center Networking (DCN): Infrastructure and Operations," *IEEE Communications Surveys Tutorials*, vol. PP, no. 99, pp. 1–1, 2016.
19. "eTorii (GitHub)." [Online]. Available: https://github.com/gistnetserv-uah/eTorii
20. J. Farkas, J. Farkas, M. Salvador, and G. dos Santos, "Automatic Discovery of Physical Topology in Ethernet Networks," in *Advanced Information Networking*

and Applications, 2008. AINA 2008. 22nd International Conference on, March 2008, pp. 848–854.
21. K. Chen, C. Guo, H. Wu, J. Yuan, Z. Feng, Y. Chen, S. Lu, and W. Wu, “DAC: Generic and Automatic Address Configuration for Data Center Networks,” *Networking, IEEE/ACM Transactions on*, vol. 20, no. 1, pp. 84–99, Feb 2012.
22. G. Deng, H. Wang, and Z. Gong, “Tree-conf: A fast automatic address configuration method for tree-like data center networks,” in *Computing, Communication and Networking Technologies (ICCCNT), 2014 International Conference on*, July 2014, pp. 1–6.
23. Y. Hu, M. Zhu, Y. Xia, K. Chen, and Y. Luo, “GARDEN: Generic Addressing and Routing for Data Center Networks,” in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, June 2012, pp. 107–114.
24. A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, “A general approach to network configuration analysis,” *Networked Systems Design and Implementation*, 2015.
25. X. Ma, C. Hu, K. Chen, C. Zhang, H. Zhang, K. Zheng, Y. Chen, and X. Sun, “Error Tolerant Address Configuration for Data Center Networks with Malfunctioning Devices,” in *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, June 2012, pp. 708–717.
26. C. Jiang, W. Liang, M. Xu, and L. Liu, “MTR: Fault tolerant routing in Clos data center network with miswiring links,” in *Local Metropolitan Area Networks (LANMAN), 2014 IEEE 20th International Workshop on*, May 2014, pp. 1–6.