http://dx.doi.org/10.5755/j01.eee ELEKTRONIKA IR ELEKTROTECHNIKA, ISSN 1392-1215, VOL. XX, NO. X, 20XX

Modelling VM Migration in a Fog Computing Environment

Pedro Juan Roig[1,2], Salvador Alcaraz[1], Katja Gilly[1], Carlos Juiz[2]
[1]*Department of Physics and Computer Architecture, Miguel Hernández University, Avda. Universidad, s/n - 03202 Elche (Alicante), Spain*
[2]*Department of Computer Science, University of the Balearic Islands, Ctra. Valldemossa, km 7.5 - 07122 Palma de Mallorca, Spain*
pedro.roig@goumh.umh.es, salcaraz@umh.es, katya@umh.es, cjuiz@uib.es

***Abstract*—Fog Computing was created to efficiently store and access data without the limitations challenging Cloud Computing deployments, such as network latency or bandwidth constraints. This is achieved by performing most of the processing on servers located as close as possible to where data is being collected. When mobile devices are equipped with limited resources and small capabilities, it would be convenient to make their associated computing and network resources follow them as much as possible. In this paper, migration process is studied and an algorithmic model is designed, selecting a generic Fat Tree architecture as the underlying topology, which may be useful to get a list of all devices being traversed through each of the redundant paths available.**

***Index Terms*— fog computing; migration; modelling; networking.**

I. INTRODUCTION

Fog Computing paradigm is characterised by the allocation of computing resources at the edge of the network, thus bringing the cloud computing assets closer to the end user [1]. In this context, special attention may be set on its use in Internet of Things (IoT) deployments, and particularly, in IoT moving environments [2].

Such moving IoT devices have special characteristics according to its limited computing capacity, limited battery resources and limited bandwidth [3], so a good solution for their implementation is to decouple their computing assets and move them to more resourceful facilities, powerful enough to take responsibility for a lot of IoT devices, but close enough to reduce latency and bandwidth usage [4]. Therefore, those facilities are composed of a bunch of capable servers, being able to assign a Virtual Machine (VM) to each user to cover the computing needs of each IoT device.

When talking about moving IoT devices, this outlook is crucial, as those devices lack resources of all kinds [5], such as those related before, and the use of a VM to carry the computing assets of each device may help to cope with the issues regarding resources [6].

However, a new problem arises with moving IoT devices, because as they are moving around, those VMs might end up being too far away in the complex network architecture from their associated devices, hence, those VMs should be moved as close as possible. This mechanism is called VM migration and must be taken into account in those environments [7].

Therefore, two types of movements are to be distinguished herein, this is, the movement of the device throughout the coverage area and the movement of the virtual machine associated to that device trying to get as close as possible to its owner.

The first sort of movement, this is, the one regarding just the moving IoT device is called mobility and its study is all about trying to model the most usual movements as well as the not so usual ones. Regarding literature, there have been some attempts to model general human movements in wireless environments, such as [8] and [9]. There are also some simple mathematical models, such as the ones proposed in [10] and [11], and other more complex models related to crowd interaction, such as [12], [13] and [14].

The second kind of movement, this is, the one related to VMs associated to moving IoT devices trying to follow them around, brings about the issue of trying to migrate a VM from the server hosting it to another one being located nearer to the actual position of the moving IoT device in order to facilitate the interaction between the device and its computing power, as a consequence of reducing the latency and bandwidth of such communications.

Regarding literature, a conceptual live VM migration framework is proposed in [15], also agreed in [16] for Cloud Computing and in [17] for Fog Computing, whereas a comparison between live VM migration in both environments for multimedia services is presented in [18].

In this paper, we are going to focus on studying the VM migration happening in such situations and furthermore getting a general algorithm for modelling it in a generic Fat Tree architecture, that making an interesting framework being able to support the necessary infrastructure for allocating VMs within physical servers and facilitating VM migration throughout any pair of available servers.

The organisation of this paper will be as follows: first, Section 2 introduces a general procedure for live VM migration, then, Section 3 shows a Clos network overview,

Manuscript received April XX, 20XX; accepted April XX, 20XX.
This research was funded by a grant (No. XXX-00/0000) from the Research Council of Lithuania. This research was performed in cooperation with the Institution.

next, Section 4 describes the behaviour of a Fat Tree architecture from the modelling point of view, later, Section 5 proposes a general algorithm aimed at modelling VM migration in a Fat Tree topology, after that, Section 6 presents a method for getting all devices on redundant paths, and finally, Section 7 will draw the final conclusions.

II. Live VM Migration Process

Regarding VM migration, there are three main approaches to be taken [19], such as cold migration, where the VM is shut down before moving it, hot migration, where just its OS is suspended before the movement, and finally live migration, thus allowing the services running on it to be keep going in a seamless manner whilst the movement is performed, that being the most interesting situation.

There are three key parameters to measure the performance of live migration [20], such as downtime, representing the amount of time the VM is halted during the migration, total migration time, carrying the amount of time elapsed for the whole process, and the amount of dirty pages migrated, referring to the data being changed during the process, and therefore, having to be further sent over again.

Regarding the live VM migration process, a tradeoff must be considered between downtime and total migration time. In order to achieve this, the memory transfer is the key player, although connections to local devices and network interfaces may also be taken into account.

Generally speaking, memory transfers may be broken up into three stages [21], such as push phase, where source VM keeps running whilst the transfer process starts taking place, stop-and-copy phase, where source VM is halted, pages are copied through and, in turn, destination VM is started, and pull phase, where the new VM runs and if a requested page has not yet being copied, it is retrieved from the source VM.

Based on the above, some techniques have been proposed to undertake the live VM migration process in an efficient manner by just focusing on one or two of the stages described above, such as pure stop-and-copy, pure demand-migration or post-copy live migration. However, it seems that the most efficient approach is the pre-copy migration, composed by a combination of a bounded iterative push stage with a very short stop-and-copy stage, where a number of iterations take place until all dirty pages have already been transferred.

The pre-copy migration process between two hosts may be divided into some six stages, where a VM transaction between any two hosts takes place, according to a pre-established migration timeline:

1. Pre-migration, where a destination host with enough resources is preselected
2. Reservation, where resources are allocated beforehand at that destination host
3. Iterative pre-copy, where the whole RAM is sent in the first iteration, and dirty pages are sent in the following iterations
4. Stop-and-copy, where the source VM is halted so as to copy its CPU state and remaining inconsistent pages to the destination VM
5. Commitment, where destination host acknowledges it has received a consistent VM copy and the source host acknowledges it back prior to discarding the original VM
6. Activation, where the migrated VM gets activated and device drivers are attached to the new VM

To round it all up, Figure 1 exhibits the timeline for the live VM iterative pre-copy migration process.

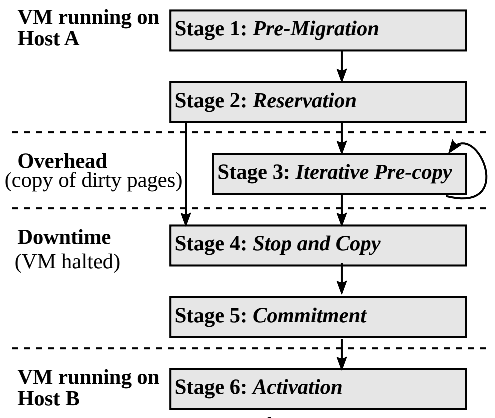

Fig. 1. VM Migration Timeline

III. Clos Networks

Back in the fifties, Clos networks were designed in order to switch telephone calls in an efficient manner [22] by virtue of using crossbar switches. Basically, the point was the use of equipment with multiple stages of interconnection in order for the calls to be completed, hence providing alternative paths between sources and destinations, thus allowing the phone call to be always connected and not blocked by any other call.

Later in the nineties, Ethernet switches came along and the concept of Clos networks was expanded so as to achieve cost-effective, reduced operational complexity and limited scalability [23]. The point there was to create multistage topologies built with commodity switches, so cost-effective deployments might be attained.

Afterwards, with the arrival of the 21st century, Data Centers and Cloud Computing facilities are still making use of those topologies, with different proposals such as two-stage designs [24], three-stage ones [25], or even alternative ones [26], each one having its own benefits and drawbacks, hence providing a full range of solutions in order to deal with different situations.

Those topologies may well be used regarding the underlying structure of Fog Computing environments, in order to host VMs and support the necessary live VM migrations, where two of the main proposals in literature are Leaf and Spine [27] and Fat Tree architectures [28].

Leaf and Spine is a 2-tier topology, where the lower one is composed by switches directly connecting with servers, and the upper one is made of switches interconnecting the lower ones in a full mesh fashion. This design provides full redundancy, as there are always a number of redundant paths among any two given switches which is equal to the number of switches being part of the upper layer. However, that is its main drawback, as it is prone to scalability issues as the number of switches gets increased, and so is the number of redundant connections to be provided. Figure 2 depicts an example of such topology with 4 switches in the Spine layer.

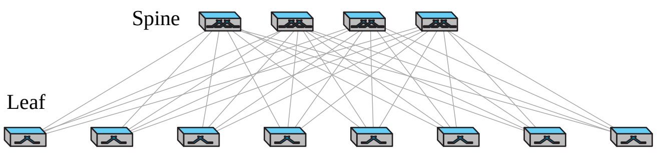

Fig. 2. Leaf and Spine topology

On the other hand, Fat Tree is an alternative to the above, allowing better scalability. It is a 3-tier topology, subdivided in Pods, where there is full mesh interconnection among the switches located in the lower layer and the upper layer of each Pod, such as the above case, but there is an extra top layer which is in charge of interconnecting the different Pods taking part of the topology.

This way, there are less redundant paths among any two given switches, but there are no scalability issues any more. Figure 3 exhibits an example of that topology with K=4 and 1:1 oversubscription, meaning that all theoretical links in the topology have been used.

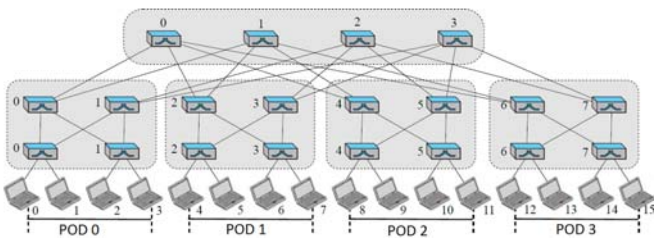

Fig. 3. Fat Tree topology

IV. FAT TREE BEHAVIOUR FOR MODELLING

In order to obtain a modelling for the Fat Tree architecture, it is necessary to describe how each layer of the topology behaves. To start with, taking the Fat Tree architecture depicted above as a ground for a formal model, devices have been identified from left to right for each different layer so as to design a model showing how this architecture allows the communication flow among servers.

Fat Tree architecture is composed by 3 layers of switches, where Edge layer is the lower one, Aggregation layer is the middle one, and Core layer is the upper one. Additionally, a bottom layer holding hosts, or servers, is also considered.

Taking the name of Fat Tree, the word “Tree” comes from the inverted tree-like structure of this architecture, where Core layer might be seen as the root layer and Hosts layer as the leaves layer.

Furthermore, following that analogy, the word “Fat” comes from the existence of more links on the root layer than in the leaves layer, such that there are $(K/2)^2$ links between upper and middle layer, $(K/2)^1$ links between middle and lower layer, and $(K/2)^0$, this is, 1 link between lower and hosts layer.

The Fat Tree structure may be considered as a K-ary tree, being K the main parameter of this structure, as there are K Pods, each of those containing K switches, divided into lower and middle layer, and also each switch has K ports.

In Figure 3 exhibited above, K=4 has been used, although it may be extended to any natural even number, and as such, it will be represented by the algorithmic model to be shown.

Therefore, the model might be built up by looking at Figure 3 exhibited above as a reference. As it may be seen over there, switches at the all layers have been numbered from 0 onwards, considering left to right direction.

As a matter of fact, there is a total of (K/2) hosts hanging out of each lower switch, which means that there are $(K^2/2)$ hosts hanging out of each Pod, which also means that there are $(K^3/4)$ hosts in the whole topology. In addition to that, there is a total of $(K^2/2)$ switches in both the lower layer and the middle layer, whereas there are $(K^2/4)$ in the upper one.

As per the nomenclature of the ports of each item, it is to be said that the servers only have one port, which will be called as 0, whereas on the switches the ports will be named from 0 to (K-1) from left to right, starting from the bottom and finishing on the top. All this is represented in Figure 4.

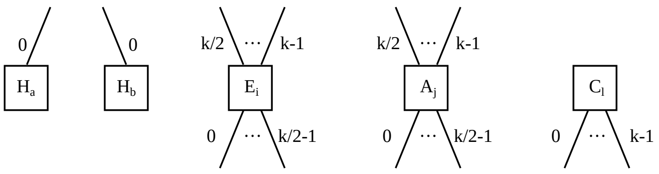

Fig. 4. Links on each layer of the Fat Tree architecture

With all this in mind, the model will state the atomic actions for communicating messages involved in the live VM migration process, such as send or receive, in a way that both will bear the item and the corresponding port taking part in such communication. Furthermore, the model will show the decision-making processes in order to guide a VM from a given source to the proper destination, following the optimum paths, that being one, two or three hops away.

The arguments for the send and receive actions are the source host (a), the destination host (b) and the proper VM to be migrated owned by a user identifier (u), this is, VM(u).

All items within each layer are modelled in a generic manner, such as Host H_h, Edge E_i, Aggregation A_j and Core C_l, where each variable is bounded by the number of items for each layer exposed above. The model has been expressed with some snippets coded in C-style for clarity purposes, and it takes into account all considerations made.

V. VM MIGRATION MODEL

- *HOST H_h:*

The server layer, also known as Host layer, is the easiest to model, as each server (h) may perform just two actions, such as receiving or sending a VM, those being the key actions chosen herein for modelling the VM migration process. Further considerations such as VM creation or VM termination are not borne in mind for simplification purposes. Let us also suppose there is enough room to allocate the VMs assigned to all the users within the system.

Therefore, all the servers in the topology are awaiting to undertake any of those two actions at any given time, at it is not usually known beforehand when a VM migration is going to take place. As per the receiving part, it is performed when the VM associated to a particular user is not located into a given host, hence, the host is ready to receive that VM in anytime. Otherwise, the sending part is carried out when the aforesaid VM is indeed located in that host and it is time to leave. This is shown in Algorithm 1.

Additionally, each server has only one port, namely eth0, so messages to and from the Edge layer move through that single port, whilst messages have the structure (a,b,VM(u)).

```
for (h = 0; h < (K^3/4); h++)
{
 while (1)
 {
  for (u = 0; u < TOTAL_USERS; u++)
  {
   if NOT (VM(u) in h)
    receive_{HOST {h}, PORT {eth0}} (a,h,VM(u));
   else if LEAVE_NOW(VM(u))
    send_{HOST {h}, PORT {eth0}} (h,b,VM(u));
   else
    STAY_IN(VM(u));
   }
  }
 }
}
```

Algorithm 1. HOST( )

- *EDGE E_i:*

The lower layer, also known as Edge layer, is the one directly connected to the servers. Each switch located therein is continuously monitoring all its ports in order to receive a VM to be translated from a source host to a destination host.

It is to be remarked that any switch located in this layer has the lower half of its ports looking downwards, whereas the upper half are looking upwards.

On the one hand, in case a VM is received from a given host on any switch, it will be coming from any of its lower ports, and then, two cases may be distinguished. If the destination is another host hanging on the same switch, it will be forwarded downwards straight to that other host. In this case, both hosts may be considered as being part of the same IntraNet, as they may be just one-hop away to each other. Otherwise, if that is not the case, it will be forwarded upwards.

On the other hand, in case a VM is received from any switch standing on the middle layer, it will be coming from any of its upper ports, and then, it will be guided through the port directly connected to the destination host. This all may be observed in Algorithm 2.

```
for (i = 0; i < (K^2/2); i++)
{
 while (1)
 {
  for (m = 0; m < (K/2); m++) {
   if(receive_{EDGE {i}, PORT {m}} (a,b,VM(u))){
    if (int[a/(K/2)] == int[b/(K/2)])
     send_{EDGE {i}, PORT {b mod (k/2)}} (a,b,VM(u));
    else
     for (m′ =(K/2); m′ < K; m′++)
      send_{EDGE {i}, PORT {m′}} (a,b,VM(u));
   }
  }
  for (m′ = (K/2); m′ < K; m′++) {
   if(receive_{EDGE {i}, PORT {m′}} (a,b,VM(u))){
    send_{EDGE {i}, PORT {b mod (k/2)}} (a,b,VM(u));
  }
 }
}
```

Algorithm 2. EDGE( )

- *AGGREGATION A_j:*

The middle layer, also known as Aggregation layer, has the same port configuration as the lower layer. It is to be noted that there is a full mesh topology among switches staying on both layers within a single Pod.

As in the previous layer, all switches therein are monitoring its ports all the time waiting for incoming VMs. Therefore, if a VM is received from any lower ports of any of these switches, two case scenarios may be distinguished.

If the destination is another host situated on the same Pod, that VM will be forwarded downwards through the lower switch where the destination is hanging on, as both source and destination hosts may be considered as being part of the same Pod, also known as IntraPod, this is, both being two-hops away to each other. Otherwise, if this is not the case, it will be forwarded upwards.

Alternatively, if a VM is received from any upper ports of any of those switches, it will be headed for the lower layer switch where the destination host is hanging on. This all is displayed in Algorithm 3.

```
for (j = 0; j < (K^2/2); j++)
{
 while (1)
 {
  for (n = 0; n < (K/2); n++) {
   if (receive_{AGGR {j}, PORT {n}} (a,b,VM(u))){
    if (int[a/(K/2)^2] == int[b/(K/2)^2])
     send_{AGGR {j}, PORT {int[b/(k/2)]}} (a,b,VM(u));
    else
     for (n′ = (K/2); n′ < K; n′++)
      send_{AGGR {j}, PORT {n′}} (a,b,VM(u));
   }
  }
  for (n′ = (K/2); n′ < K; n′++) {
   if(receive_{AGGR {j}, PORT {n′}} (a,b,VM(u))){
    send_{AGGR {j}, PORT {int[b/(k/2)]}} (a,b,VM(u));
  }
 }
}
```

Algorithm 3. AGGREGATION( )

- *CORE C_l:*

The upper layer, also known as Core layer, is the one interconnecting the different Pods, so the ports of all its switches are looking downwards, providing a full mesh topology among all the existing Pods.

Therefore, all of switches are waiting to receive a VM through any of its corresponding ports, and when that happens, it is redirected to its directly connected middle layer switch on the Pod holding the destination host.

In this case, source and destination hosts do not share the same Pod, also known as InterPod, meaning both are three-hops away to each other. This all is shown in Algorithm 4.

```
for (l = 0; l < (K^2/4); l++)
{
 while (1)
 {
  for (p = 0; p < K; p++) {
   if(receive_{CORE {l}, PORT {p}} (a,b,VM(u))){
    send_{CORE {l}, PORT {int[b/(k/2)^2]}} (a,b,VM(u));
  }
 }
}
```

Algorithm 4. CORE( )

VI. GETTING ALL DEVICES ON REDUNDANT PATHS

All the above may be used in order to get a list of devices for each of the redundant paths taken from a given source host to a given destination host, as it may be seen in

Algorithm 5. The nomenclature for the devices is the same one used so far.

- *LIST OF DEVICES:*

```
DeviceList(a,b){
 items = [];
 items += [ $H_a$ ];
 if (int[a/(K/2)] == int[b/(K/2)])
 {
  items += [ $E_{int[a/(K/2)]}$ $H_b$ ];
  t = 1;
  topology = “INTRANET”;
 }
 else if (int[$a/(K/2)^2$] == int[$b/(K/2)^2$])
 {
  items += [ $E_{int[a/(K/2)]}$ “(“ ];
  for (q = (K/2) · int[$a/(K/2)^2$];
       q < K/2) · int[$a/(K/2)^2$] + (K/2); q++)
   items += [ $A_q$ ];
  items += [ “)” $E_{int[b/(K/2)]}$ $H_b$ ];
  t = 2;
  topology = “INTRAPOD”;
 }
 else
 {
  items += [ $E_{int[a/(K/2)]}$ “(“ ];
  for (q = (K/2) · int[$a/(K/2)^2$];
       q < K/2) · int[$a/(K/2)^2$] + (K/2); q++)
  {
   items += [ $A_q$ “{“ ];
   for (r = q · (K/2); r < q · k; r++)
    items += [ $C_r$ ];
   s = (K/2) · int[$b/(K/2)^2$]  + r mod (K/2) ;

   items += [ “}” $A_s$  ];
  }
  items += [ “)”$E_{int[b/(K/2)]}$ $H_b$ ];
  t = 3;
  topology = “INTERPOD”;
 }
 print(“Topology: %s\n”,topology);
 print(“Number of Hops: %d\n”,t);
 print(“Redundant Paths: %d\n”,(K/2)^(t-1));
 print(“Items: %s\n”, items);
}
```

Algorithm 5. DeviceList(a,b)

The list presented above may be extended with the corresponding ports used in each link for all redundant paths, as it may be seen in Algorithm 6. The ports will be expressed within parenthesis, attached to its corresponding device with the sign “*”, and a link between two ports will be expressed by the signs “---”, appearing in between both ends of such a link.

- *LIST OF DEVICES AND PORTS:*

```
DeviceAndPortsList(a,b){
 if (int[a/(K/2)] == int[b/(K/2)])
 {
  t = 1;
  topology = “INTRANET”;
 }
 else if (int[$a/(K/2)^2$] == int[$b/(K/2)^2$])
 {
  t = 2;
  topology = “INTRAPOD”;
 }
 else
 {
  t = 3;
  topology = “INTERPOD”;
 }
 print(“Topology: %s\n”,topology);
 print(“Number of Hops: %d\n”,t);
 print(“Redundant Paths: %d\n”,(K/2)^(t-1));
 // REDUNDANT PATHS
 if (topology == “INTRANET”)
 {
  Path(0) = [ $H_a$ *
          * (0) --- (a mod (K/2))*
          * $E_{int[a/(K/2)]}$ *
          * (b mod (K/2)) --- (0) *
          * $H_b$ ];
 }
 else if (topology = “INTRAPOD”)
 {
  x = 0;
  for (q = (K/2) · int[$a/(K/2)^2$];
       q < K/2) · int[$a/(K/2)^2$] + (K/2); q++)
  {
   Path(x) = [ $H_a$ *
           * (0) --- (a mod (K/2))*
           * $E_{int[a/(K/2)]}$ *
           * ((K/2) + x) --- (int[a/(K/2)])*
           * $A_q$ *
           * (int[b/(K/2)]) --- ((K/2) + x) *
           * $E_{int[b/(K/2)]}$ *
           * (b mod (K/2)) --- (0) *
           * $H_b$];
   x++;
  }
 }
else  // if(topology == “INTERPOD”)
 {
  y = 0;
  for (q = (K/2) · int[$a/(K/2)^2$];
       q < K/2) · int[$a/(K/2)^2$] + (K/2); q++)
  {
   z = 0;
   for (r = q · (K/2); r < q · K; r++)
   {
    s = (K/2) · int[$b/(K/2)^2$] + r ·mod (K/2);
    Path(y · (K/2) + z) =
         = [ $H_a$ *
           * (0) --- (a mod (K/2))*
           * $E_{int[a/(K/2)]}$ *
           * ((K/2) + y) --- (int[a/(K/2)])*
           * $A_q$ *
           * ((K/2) + z mod (K/2)) ---
             --- (int[$a/(K/2)^2$]) *
           * $C_r$
           * (int[$b/(K/2)^2$]) ---
             --- ((K/2) + z mod (K/2)) *
           * $A_s$ *
           * (int[b/(K/2)]) --- ((K/2) + y) *
           * $E_{int[b/(K/2)]}$ *
           * (b mod (K/2)) --- (0) *
           * $H_b$];
    z++;
   }
   y++;
  }
 }
 // SUMMARY OF PATHS
 for (c = 0; c < (K/2)^(t-1); c++)
  print(“Path(%d) = %s\n”,c,Path(c));
}
```

Algorithm 6. DeviceAndPortsList(a,b)

Regarding evaluation and verification of the VM migration algorithms proposed, some executions showing all case scenarios may do it, thus considering source and destination Hosts being 1-hop away, 2-hops away or 3-hops away, with a generic K. Let us focus on the algorithm DeviceList, as the algorithm DeviceAndPortsList is just an extension of the former showing the port identifiers involved for each device.

First of all, let us take a scenario being IntraNet. The model considers both Hosts being 1-hop away, with just 1 path between them, as there is just a single Edge switch defining the only path for any pair of Hosts hanging out of it. Therefore, the first conditional sentence in the algorithm will hold and that Edge switch is going to be identified.

Then, let us take an IntraPod scenario. The model considers both Hosts being 2-hops away, with K/2 redundant paths in between, as both Hosts share the same

Pod, so each Aggregation switch within that single Pod defines a different path, being reached from the same source Edge switch and being redirected to the same destination Edge switch. Thus, the second conditional sentence in the algorithm will hold and the components of each path are going to be identified.

Finally, let us take an InterPod scenario. The model considers both Hosts being 3-hops away, with $K^2/4$ redundant paths between both, as both Hosts stand in different Pods, hence, each Core switch defines a different path, being reached from one of the Aggregation switches in the source Pod and being redirected to one of the Aggregation switches in the destination Pod. Therefore, the else sentence in the conditional sentence will hold and all the components of each path are to be identified.

To round it all up, the use of this algorithm matches what happen in a real Fat Tree architecture regarding all possible case scenarios, therefore, the model may be considered as verified.

VII. CONCLUSIONS

In this paper we have been studying the VM migration process between a given source host and a given destination host, both being interconnected through a Fat Tree architecture. First of all, migration types have been exposed, noting that live VM migration is the most interesting one, and among all its variations, iterative pre-copy is the most efficient regarding the tradeoff between downtime and total migration time.

Then, Clos networks have been presented, and a comparison between Leaf and Spine and Fat Tree architectures has been introduced, leading to the consideration of Fat Tree as being more scalable, and as such, that has been select to build an algorithmic model regarding VM migration.

Eventually, each three layers of Fat Tree and an additional layer for the hosts where VMs are located have been modelled. The behaviour of each layer have been expressed in terms of receiving a VM owned by a user, coming from a source host and going to a destination host, and in turn, sending it to the optimal path, to reach that destination with the minimal possible number of hops. If that number of hops is more than one, there are redundant paths being all of them optimal, so the model gives them all.

As a sort of proof of concept, two algorithms has been proposed, the first one listing the devices to be traversed through all redundant paths available from a given source host to a given destination host, and the second one being an extension of the former by also quoting the ports involved in each link through each redundant path.

In conclusion, the algorithmic model proposed, based on arithmetic operations, succeeds in expressing the VM migration process from one host to another in a Fat Tree architecture.

REFERENCES

[1] C.S.R. Prabhu, *Overview - Fog Computing and Internet-of-Things (IoT)*, EAI Endorsed Transactions on Cloud Systems 2017, Issue 10, Article 5, pp. 1-24.

[2] M.R. Anawar, S. Wang, M.A. Zia, A.K. Jadoon, U. Akram, S. Raza, *Fog Computing: An Overview of Big IoT Data Analytics*, WCMC'2018, Article ID 7157192, pp. 1-22.

[3] R. Mahmud, K. Ramamohanarao, R. Buyya, *Latency-aware Application Module Management for Fog Computing Environments*, ACM Transactions on Internet Technology, March 2018, Vol. 1, No. 1, pp. 1-22.

[4] T. Patel, K. Jariwala, *Fog Computing in IoT*, in ARSSS International Conference'2018, Vol. 1, pp. 17-21.

[5] S. Virushabadoss, C. Bhuvaneswari, *Analysis of Behavior Profiling Algorithm to Detect Usage Anomalies in Fog Computing*, IJESI-NCIOT'2018, Vol. 1, pp. 14-19.

[6] J. Abdelaziz, M. Adda, H. MCheick, *An Architectural Model for Fog Computing*, JUSPN'2018, Vol. 10. No. 1, pp 21-25.

[7] K. Gilly, S. Filiposka, A. Mishev, *Supporting Location Transparent Services in a Mobile Edge Computing Environment*, AECE'2018, Vol.18, No.4, pp.11-22.

[8] M. Musolesi, C. Mascolo, *Mobility Models for Systems Evaluation - A Survey*, 2008.

[9] A. Hess, K.A. Hummel, W.N. Gansterer, G. Haring, *Data-driven Human Mobility Modeling: A Survey and Engineering Guidance for Mobile Networking*, in ACM Computer Survey, 2016, Vol. 48, No. 3, Art. 38, pp. 1-39.

[10] T. Camp, J. Boleng and V. Davies, *A survey of mobility models for ad hoc network research*, in WCMC'2002, Vol. 2, Issue 5, pp. 483-502.

[11] P.J. Roig, S. Alcaraz, K. Gilly, C. Juiz, *Study on Mobility and Migration in a Fog Computing Environment*, in ELECTRONICS'2018, pp. 1-6.

[12] O. Hesham, G.A. Wainer, *Centroidal particles for interactive crowd simulation*, 2016.

[13] Y. Xu, Z. Piao, S. Gao, *Encoding Crowd Interaction with Deep Neural Network for Pedestrian Trajectory Prediction*, in CVPR'2018, Vol 1, pp. 5275-5284.

[14] T. Bosse, M. Hoogendoorn, M. Klein, J.Treur, C. van der Wal, A. van Wissen, *Modelling collective decision making in groups and crowds: Integrating social*, in AAMAS'2013, Vol 27, Issue 1, pp. 52-84.

[15] O. Osanaiye, S. Chen, Z. Yan, R. Lu, K.R. Choo, M. Dlodlo, *From Cloud to Fog Computing: A Review and a Conceptual Live VM Migration Framework*, in RACRAN'2017, Vol. 5, pp. 8284-8300.

[16] P. Kaur, A. Rani, *Virtual Machine Migration in Cloud Computing*, in IJGDC'2015, Vol. 8, No. 5, pp. 337-342.

[17] Y.S. Rao, K.B. Sree, *A Review on Fog Computing : Conceptual Live Vm Migration Framework, Issues, Applications and Its Challenges*, IJSRCSEIT'2018, Vol. 3, Issue 1, pp. 1175-1184.

[18] D. Rosário et al., *Service Migration from Cloud to Multi-tier Fog Nodes for Multimedia Dissemination with QoE Support*, in Sensors 2018, Issue 2, Article 329, pp. 1-17.

[19] M. Forsman, A. Glad, L. Lundberg, D. Ilie, *Algorithms for automated live migration of virtual machines*, in JSS'2015, Vol. 101, pp. 110-126.

[20] Y. Ruan, Z. Cao, Z. Cui, *Pre-filter-copy: Efficient and self-adaptive live migration of virtual machines*, IEEE S.J.'2016, Vol. 10, No. 4, pp. 1459-1469.

[21] C. Clark et al., *Live migration of virtual machines*, in NSDI'2005, Vol. 2, pp. 273-286.

[22] C. Clos, *A study of non-blocking switching networks*, BST Journal'1953, Vol. 32, Issue 2, pp. 406-424.

[23] A. Singh et al., *Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network*, SIGCOMM'15, pp. 1-15.

[24] R. Rojas-Cessa, C. Lin, *Scalable two-stage Clos-network switch and module-first matching*, in HPSR'2006, pp. 1-6.

[25] X. Yuan, *On Nonblocking Folded-Clos Networks in Computer Communication Environments*, in IEEE IPDPS'2011, pp. 1-9.

[26] F. Hassen, L. Mhamdi, *High-Capacity Clos-Network Switch for Data Center Networks*, in IEEE ICC'2017, pp. 1-7.

[27] K.C. Okafor, *Leveraging Fog Computing for scalable IoT datacenter using Spine-Leaf network topology*, in JECE'2017, pp. 1-11.

[28] M. Al-Fares, A. Loukissas and A. Vahdat, *A scalable, commodity data center network architecture*, in SIGCOMM'2008, pp. 63-74.