Cost-efficient Allocation of Clustered Network Emulation Resources Using Feedback

Author: **Tamas Levai**

Neptun-code: **BUIHKR**

E-mail: **levait@tmit.bme.hu**

Consultant: **Felician Nemeth**

E-mail: **nemethf@tmit.bme.hu**

Budapest, 2015.

Abstract

Network emulation is a technology in which certain network elements (such as links and switches) are virtual, but, with the exact same behaviour as the physical ones. As a result of virtualizing certain network elements, the creation of fully emulated or partially emulated networks has become possible. Network emulation itself is a relatively recent and promising field of research and development with actual significance. Namely, in the last years many technologies like software defined networks (SDN), network function virtualization (NFV), and cloud computing emerged from network emulation.

Due to the increasing needs and technological improvement it is possible to emulate all kind of networks in a scale from simple virtual links to complete data centers. There is a real need in research and in production for development, testing and measuring purposes to emulate more and more complex networks. Necessarily, complex emulated networks are heavy on physical resources, therefore, it is necessary to interconnect multiple emulator nodes. But then, mapping the network to emulate onto the infrastructure is not trivial. The current algorithms generally use the bin packing problem or the integer linear programming relaxed by heuristics. As oppose to this approach, the presented algorithm tries to find the optimum by trial and error method. In every iteration it tries to allocate the network emulation resources in a subset of physical resources first. Then, it tests the success of allocation with on-line tests. The next iteration is based on the feedback from the tests. A cost efficient allocation will be the result of the algorithm.

In the work beside the introduction of existing clustered network emulation tools I present my new algorithm in detail with a proof of concept implementation using Mininet Cluster Edition and the evaluation of this implementation.

Hungarian Abstract

Hálózatemulációról beszélünk ha az egyes hálózati elemek (például linkek, kapcsolók) ugyan virtuálisak, ám viselkedésük megegyezik a fizikailag létezők viselkedésével. Az egyes hálózati elemek virtualizálására építve lehetővé vált teljesen emulált vagy csupán részlegesen emulált hálózatokat létrehozása. A hálózatemuláció viszonylag friss és ígéretes kutatási és fejlesztési irány, melynek térnyerése aktuális. Ugyanis megfigyelhető, hogy az utóbbi pár évben erre épülve olyan új paradigmát jelentő technológiák jöttek létre, mint a szoftveresen definiált hálózatok (SDN), a hálózati funkció virtualizálás (NFV) vagy felhő alapú számítástechnika (cloud computing).

A felhasználási igények és a technológiai fejlődés hatására az emulált hálózatok hatásköre jelenleg az egyszerű virtuális linkektől komplett adatközpontok emulálásáig tart. Valós igényként jelentkezik mind a kutatások terén, mind az iparban, hogy fejlesztési, tesztelési és mérési célokra egyre komplexebb hálózatokat lehessen emulálni. Természetesen a komplex emulált hálózatok erőforrás igénye jelentős, több emulációt futtató elem összekapcsolását követeli meg. Viszont az emulálandó hálózat hatékony felosztása az infrastruktúrán nem triviális feladat. Az alkalmazott algoritmusok jellemzően vagy a ládapakolási problémát vagy az egészértékű lineáris programozást veszik alapul és relaxálják azt különböző heurisztikák segítségével. Ezekkel szemben a dolgozatban bemutatott algoritmus az optimumot alulról próbálja megközelíteni próba-hiba módszer segítségével, azaz minden lépésben megpróbálja lefoglalni az emulációhoz szükséges összes erőforrást a fizikai erőforrások egy részhalmazán, majd a lefoglalás sikerességét aktív tesztekkel ellenőrzi, melyek eredményét visszacsatolva történik a következő iteráció. Ily módon futtatva az allokáló algoritmus eredménye egy költséghatékony lefoglalás.

A dolgozatban a létező elosztott hálózatemuláló szoftverek bemutatásán túl részletesen ismertetem az általam megvalósított új algoritmust, illetve annak megvalósítását igazoló Mininet Cluster Edition alapú mintaimplementációját, melynek kiértékelése is része a dolgozatnak.

Contents

1 Introduction **1**

2 Related Work **3**

3 Utilising Trial and Error Approach for Network Emulation Resource Allocation **10**

3.1 Motivation 10

3.2 Requirements 10

3.3 The Algorithm in Detail 13

3.4 Proof of Concept 17

4 Evaluation **21**

4.1 Evaluation setup 21

4.2 Results 22

5 Conclusion **27**

5.1 Summary 27

5.2 Future Work 27

6 References **28**

1 Introduction

On high-level this document focuses on providing network emulation as a service in an environment with dynamically changing resources such as cloud computing. Particularly, targeted area of this topic in this document is the allocation of network emulation resources. The presented approach targets cost-efficiency, reliability.

Addressing the problem in detail requires further clarification. Network emulation is a well-known technology with a long tradition. It provides virtual network elements that behave the same way as the non-virtual ones. It is possible to emulate all of the ISO model's layers, therefore, it is possible to virtualise by software all kinds of network elements from links to hosts even on large scale. Nowadays we can differentiate internal network virtualisation and external network virtualisation. In case of internal network virtualisation the virtualised network runs on a single machine (e.g., PC). In this scenario the hosts, interfaces and links are all virtual. There are several options to implement internal network virtualisation, one common method is to implement all the virtualisation in kernel space, and rely on these features. It is usually realised in Linux, since all of the required tools are already existing in it. Virtual network interfaces are supported since a long time ago. Virtual links can be created by the traffic controlling and shaping features of the kernel; network interfaces can be connected by virtual bridges, link quality can be set in the network scheduler by configuring tc (traffic control). The virtualised end hosts can be virtual machines (e.g., Kernel-based Virtual Machine) or software containers (e.g., Linux Containers). Utilising these basic tools by hand is cumbersome, therefore, several wrapper tools (e.g., Mininet [4]) exist to deploy and maintain virtual networks. This way it is possible to emulate large networks even with hundreds of hosts on off-the-self personal computers. This number can be increased a bit by various slightly distorting tricks such as time dilation [6]. But even with tricks, this order of magnitude is not sufficient for current research and testing purposes, such as studying traffic in data centres or in complete telecommunication networks.

External network virtualisation can provide solution for this scaling problem

by interconnecting multiple hosts. It utilises the technologies mentioned by inter network virtualisation plus the virtual links between the virtual network components running on different hosts. These virtual links are using tunnelling protocols over physical links. For this purpose several tunnelling technologies (i.e., Generic Routing Encapsulation tunnel and Secure Shell tunnel) are used. These tunnels can be connected to the virtual switches (e.g., Open vSwitch) running on the arbitrary hosts, so the internally virtualised network segments can be interconnected into a whole externally virtualised network.

Many new technologies emerged from network virtualisation. Based on these subsequent technologies (e.g., Software Defined Networking) and the significant development and utilisation of virtualisation technologies, cloud computing came into existence. The fundamental idea behind cloud computing is to decouple the running software from the underlying hardware. To achieve that the cloud computing environment consist of managed virtual machines and virtual links atop a computer cluster. As a consequence of this infrastructure, it is possible to dynamically scale the set of resources available for running applications. This approach seems a big success in research and in production too, therefore, there is a real need to develop methods to emulate networks on this platform in a cost-effective and reliable manner.

The rest of this document is organised the following way. In Section 2, I present nowadays most common network emulators (Mininet [4, 5], Maxinet [1, 2], Distributed Openflow Testbed [3]) that support external network emulation in detail. The main work, the proposed approach suiting for cloud computing environment is presented in Section 3. Evaluation results of the proof of concept are described in Section 4. Finally, I conclude in Section 5.1.

2 Related Work

This section presents relevant network emulators focusing on external network emulation. All of these emulators use different algorithms for placing the emulated network elements onto the hosts.

Mininet

Mininet [4] is the de-facto network emulator, initially started as an SDN experimenting tool, that can emulate links, switches, routers, middleboxes and end-hosts. The end-host behave just like real hosts: they are able to run arbitrary programs, and they have virtual Ethernet interfaces. The links in Mininet can be characterised (e.g., bandwidth, delay) and network elements in between the end-hosts behave just like real Ethernet devices: they process the incoming data just like regular Ethernet devices. Therefore, results of measurements done in a Mininet network should look like results of measurements done in an identically same physical network.

Mininet is written mostly in Python. It provides an extensive API to describe custom network topologies and functions. Mininet currently runs only on Linux due to its dependency on underlying Linux technologies; it uses process groups, CPU bandwidth isolation, virtual Ethernet links, network namespaces and link schedulers to emulate networks. Namely, end-hosts are isolated user-level processes put into network namespaces with exclusive network interfaces, ports and routing tables. The emulated links are virtual Ethernet pairs configured by Linux Traffic Control (tc). Switches can use Linux virtual bridge or Open vSwitch to switch packets. Moreover, Mininet networks can be controlled via OpenFlow. Since version 2.2.0 Mininet also has an experimental external network emulation support via clustering Mininet instances running on arbitrary hosts.

The cluster support in Mininet heavily relies Secure Shell (SSH) and SSH tunnels, therefore, it requires some manual setup before using it. First of all Mininet must have installed on each host. The user that runs Mininet must have the same name on all machine. The hosts must have pre-configured password-less sudo and

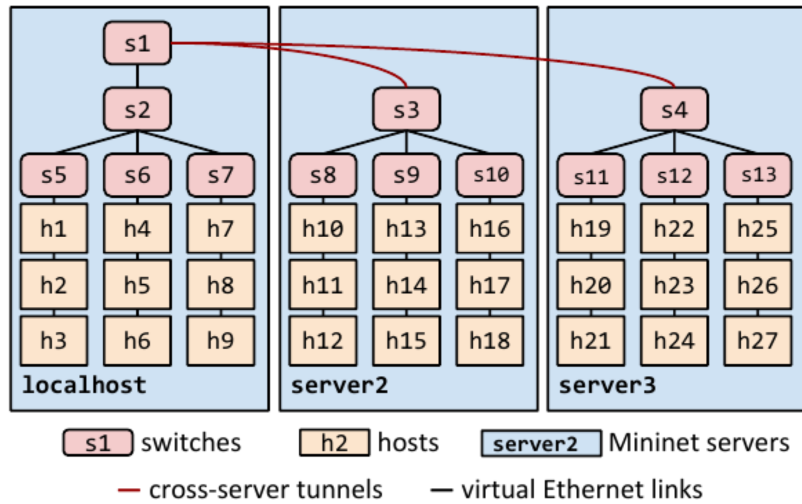

Figure 1. Mininet clustered fat-tree topology. [5]

password-less ssh access. Also, permitting tunnelling, disabling DNS, increasing the number of simultaneous sessions and allowing TCP forwarding, must be configured for the secure shell daemon on each host in order to run the clustered emulation. To ease this cumbersome process, Mininet provides a partial setup script which configures the password-less ssh access temporarily or permanently on the hosts.

On cluster start-up every arbitrary machine of the cluster starts a Mininet instance that communicates with the initiator head Mininet instance in a star topology. Even remote links are connected to this head machine. The remote links that interconnect switches on different hosts are Secure Shell tunnels as you can see in Figure 1. As a consequence, links between hosts has significant delay and limited bandwidth due to the characteristics of the underlying physical link and the overhead of SSH tunnel.

Various placement algorithms are supported in Mininet: user-defined, random that instantiates network elements on a random host of the cluster; round robin, that cycles through hosts and puts network elements one by one, and equal sized bins that solves a relaxed bin packaging problem in which the bins are the hosts and the items are the network elements, it finishes in polynomial time due to the

fix and known number of hosts. As a conclusion, we can say that all of Mininet's algorithms require to know the number of hosts inside the cluster. Moreover, for the placing decision the algorithms do not take account of the requirements of the network elements and the topology currently.

MaxiNet

MaxiNet [1, 2] is an external network emulation tool based on Mininet. It was mainly developed to experiment with novel protocols and routing algorithms in real life scenarios, especially in data centres that are emulated.

The schematic architecture of MaxiNet can be seen in Figure 2. On the bottom, the workers are computational nodes of the cluster emulating a segment of the network emulated by unmodified Mininet instances. This instances are controlled and managed from the front-end which is a special computational node in the cluster. It is the head of the architecture. Maxinet is focusing on running experiments on emulated clusters, therefore, on the top-level it contains the description of the experiment to perform on the emulated network. This experiment can be written in Python using the Maxinet API via itsets up, controls and stops the virtual network of MaxiNet. This API is very similar to Mininet's API. MaxiNet communicates with the Mininet instances via remote procedure call. The virtual links between the network segments emulated by arbitrary workers are GRE tunnels. Maxinet is also able to monitor its network and plot the statistics after the experiment is finished.

The placing algorithm of MaxiNet relies on the METIS [1] graph partitioning library. For N workers it computes N partitions of near equal weight. The optimisation criteria is minimal edge cut, mainly because, the partitioning process tries to keep most of the emulated elements locally, circumventing the limited links between hosts. Edge weights are proportional to the specific bandwidth limits of the links, and node weights are proportional to the node degree, because nodes with higher number of links are more likely cause big system load.

[1] METIS webpage: `http://glaros.dtc.umn.edu/gkhome/metis/metis/overview`

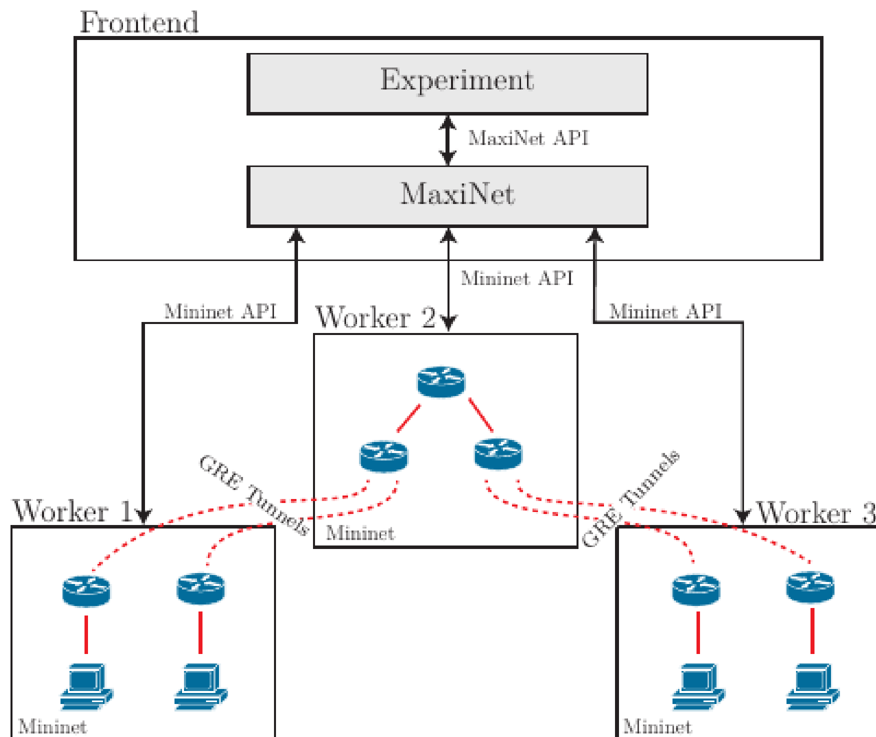

Figure 2. MaxiNet in nutshell. [1]

Distributed Openflow Testbed

Distributed Openflow Testbed (DOT) [3] is an OpenFlow compatible external network emulator. It was mainly designed to solve Mininet's scaling out issues. DOT is built from scratch using various existing technologies. In DOT Hosts are emulated by user supplied virtual machines. Switches are Open vSwitch instances. Links are either Linux virtual Ethernet pairs if both of their endpoints is on the same host, or GRE tunnels it links are interconnecting physical hosts. The link characteristics are set by Linux Traffic Control (tc). Propagation delays are set by the netem kernel module. Moreover, DOT can also provide guaranteed bandwidth between the hosts and switches.

The infrastructure of DOT consists of multiple entities which can be seen in Figure 3. The main entity is the DOT Central Manager which is responsible for allocating resources required for the network described by the user. It has two

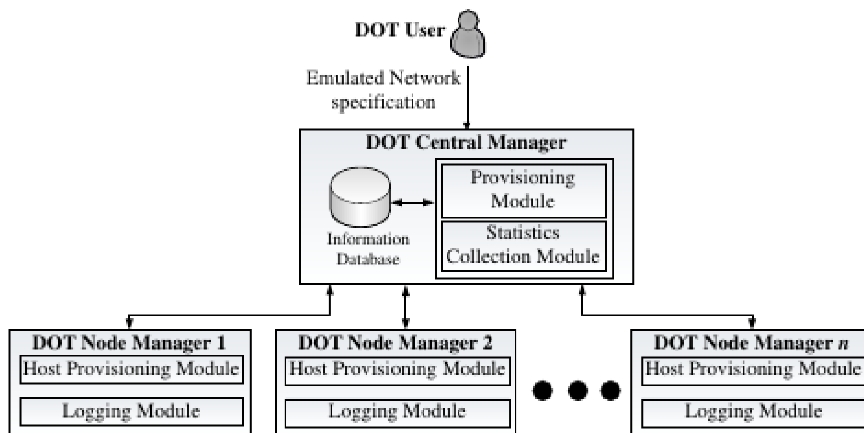

Figure 3. DOT infrastructural architecture. [3]

processing components and a storage unit: one processing component is the Provisioning Module which is responsible for running the placement algorithm, and the other is Statistics Collection Module which activates only after the placement is done. It gathers information from the logging modules placed on the nodes. The gathered information with other, management information are stored in the Information Database.

On each host there is a DOT Manager which consists of the Host Provisioning module which is responsible for allocating and configuring the required resources on that node, and the Logging module which collects local statistics: among others resource utilisation, packet rate, throughput, delay, packet loss and OpenFlow messages. Each node also runs the following software: a hypervisor to manage virtual machines and their virtual connections, Virtual machines that can act as end-hosts or middleboxes. There are also Open vSwitches as programmable OpenFlow compatible virtual switches for connecting inner links, and there is one Gateway Switch per host for handling external links. Gateway switches are transparent for DOT users, however, they require unique setup in the background. Its setup is done in three steps: first, DOT creates as many virtual Ethernet pairs as the number of virtual switches on the host, and connects the virtual switches to the gateway switch with them. The second step is the creation of the GRE tun-

nels between the physical hosts. Each virtual link has an unique identifier and the packages forwarded through them are tagged with the corresponding identifier. As a consequence of this phenomena, it is possible to identify the source Gateway Switch on the other end. Finally, the Gateway Switch has to be configured with static routes to interconnect the emulated network segments on the hosts.

The placement algorithm is the key part of the Distributed Openflow Testbed. The work of the placement algorithm is an useful example of utilising integer linear programming and heuristics for network mapping effectively. The algorithm abstracts the network as an undirected graph with virtual switches as nodes and links as edges. To find the optimal placement, the algorithm reckons multiple resource types required for the virtual hosts (e.g., CPU, memory and disk) and for the virtual links (e.g., bandwidth, propagation delay) with some relaxation: it supposes that each pair of hosts are interconnected with full bisection bandwidth. In the model, virtual switch resource requirements depend on network utilisation which correlates to the number of connecting links. Virtual machines are more complicated than virtual switches, becuase their resource requirements consist of their resource requirements of CPU and memory plus the resources required by the its connecting switch. Gateway Switches must be instantiated on each host and the resource requirements of these switches are proportional to the bandwidth capabilities of their virtual links, therefore, their resource requirements can be also estimated. Of course for the purpose of effective network emulation the best is to minimise the usage of physical links due to their translation overhead which is also a subject to the objective function used by the DOT placement algorithm. The objective function of the integer linear programming is to minimise the number of required physical nodes. This is an NP-hard problem because it generalises the multi-dimensional bin-packing problem. Therefore, a heuristic approach is used to solve this problem in polynomial time. The heuristic algorithm first places switches one by one onto active hosts based on their requirements. If there is not enough resource for a switch on any of the computational node, the algorithm activates an another host and deploys the switch there. If it is not feasible to place all the elements onto the whole set of available computational nodes, the algorithms

stops and rejects the placement. The most important heuristic used during the run is the ranking of switches based on their number of connecting links; as it was already mentioned, the resource requirement of switches are proportional to the number of their links, and also at the beginning of the algorithm's run there are a lot of unused resources, so it is easier to place those resource-heavy switches then as at later time. On the other hand switches that have neighbours already placed should be placed with high priority in order to minimise traffic between physical hosts. The other important aspect of the algorithm is the physical host selection which is based on two criteria; first, the minimisation of residual resources is important to keep resource fragmentation low and machine utilisation high. With these optimisations the algorithm's complexity is $\mathcal{O}(|N^2|log|N|+|N||H|log|H|)$, where N is the number of virtual switches, and H is the number of active computational nodes.

Winding up Distributed Openflow Testbed, one can say that it has a powerful placing algorithm and an infrastructure that is not as powerful as Mininet's infrastructure due to the overhead of the virtual machine approach used for hosts.

3 Utilising Trial and Error Approach for Network Emulation Resource Allocation

3.1 Motivation

The technologies presented in Section 2 came into existence due to the increasing need for experimenting with large networks having such big resource requirements that hardly if even could be accomplished by a single computational node. The large number of these tools forecast an increasing need for emulating large scale networks in the not so far future. An other relevant tendency of our days is the bloom (massive utilisation) of clouds mainly due to their flexibility and low costs. Cloud computing seems a promising technology for network emulation purposes too. The main advantage of cloud based network emulation is that it is really easy to use it as an end-user, and it is easier and cheaper to maintain it as system administrator than utilising dedicated servers for experiments done in large scale emulated networks.

A network emulator that suits for cloud computing environment requires great flexibility and low resource utilisation to cope with clouds clustered infrastructure. The cornerstone of these needs is the cost-efficient and reliable allocation of resources respecting the elastic nature of clouds. For this purpose, the good old trial and error approach with some feedback seems a good idea.

3.2 Requirements

The trial and error approach can guarantee minimal or almost minimal resource utilisation, but it can not guarantee reliable operation which is required for providing valid measurement results. To achieve reliability some feedback is required from the running network emulation.

The effective feedback generation relies on monitoring. Two sets of phenomenons have to be observed. One of them is the computational resource utilisation on the arbitrary hosts. Because, if there is not enough free resource on the host, the results of measurements done on the emulated network might get dis-

torted due to the degraded network performance. For network emulation the CPU usage and the memory usage are the most relevant, but disk and other I/O usage might be also suitable for detecting bottlenecks. As an example, if there is no free processor capacity, it is problematic for the emulation in the sense of validity: processing time on individual network elements might get prolonged due to the time spent waiting for the scheduler.

There are diverse methods to measure various system resources. Modern operating systems of our days log detailed statistics about the resourcses used by the system and by the individual processes. As an example, on Linux maximal-, minimal- and actual frequency, load average, time spent in userspace, time spent in kernel mode, time spent witing for I/O, CPU clocks used by the individual processes are among others logged about the central processing unit. Since different operating systems might interpret these values differently, one have to take this phenomena to account during the implementation of a multi-platform network emulator.

The other relevant metric is memory utilisation which is a wide-ranging metric; the amount of actively used, buffered, cached and swap memory can be queried. For the emulation's point of view, memory statistics are the most important at allocation time: whether the network element can be placed on a specific node or not. However, a sufficient amount of memory is required during the run of the emulation if the memory usage of the virtual hosts and middleboxes are not limited; in that case the system can run out of memory during emulation. In addition, memory consumption of background and non emulation-related software can be problematic too if they suddenly try to allocate memory (e.g., a cron process starts). If there is no more free memory available on the system, it is not guaranteed that the experiment executed on the emulated network will finish with no disturbance.

Monitoring these important system resources can be done in various ways. Fundamentally, the statistics already mentioned and even more low-level, raw statistics can be queried from the operating system, but, the low-level, raw statistics in many cases can not provide enough and relevant information. This nuisance

can be ironed out by using one of the numerous resource monitoring tools which provide trustworthy information that are easy to process and use. An implementation of the algorithm presented in Section 3.3 can rely on the information provided by these tools (e.g., top, cAdvisor).

The other phenomena to monitor is link utilisation. It is trivial that link must exist between the nodes, but it is non-trivial to discover and monitor the capabilities and changes of these links.The underpinnings of these problem are the non-trivial definitions of links: links exist on physical layer as well as on virtual layer loosely coupled. Virtual links between nodes are essentially tunnels over physical links. Tunnels are just encapsulating packets into a tunnelling packet which is then sent over the physical link. Due to the headers of this encapsulation, the packet on the virtual link has a shorter maximum transmission unit (MTU) than the packet on the wire. Therefore, these virtual links have a smaller MTU. As a consequence, the fragmentation of a packet on a virtual link is different than the fragmentation of the very same packet on the physical link. Therefore, if an error occurs, the damaged part of the packet was different on the virtual link than on the physical link. According to this phenomena, the virtual links and the physical links require different fault detection, however virtual links depend on the underlying physical links, therefore errors on the physical link can cause errors on the virtual link as well and it is merely impossible to detect the same error in the same way on both links.

If there is support from the operating system about virtual interface statistics, it is easy to monitor most of the characteristics of a link. The number of transmitted, received, dropped packets can be easily queried. If that is not the case, it is hard to detect failures like package dropping precisely. There are multiple causes of packet dropping: for example if there is not enough processing capacity, so the processing queues are overflowing or there is an issue with the link. A good approach for link error monitoring is to provide a small margin of error.

Monitoring link performance is also problematic due to the on-line behaviour of the measurement: packets must be sent on the link to calculate round trip time, practical maximum bandwidth or delay. The problem is that by sending packets

on the same link used by the emulation, the two traffic can interfere. That is not the case if we measure the link characteristics before the emulation starts, but, links characteristics in packet switched network are not constant, therefore constant monitoring is required which unfortunately utilises these links.

Beside the difficulty of measuring link metrics, it is really easy to validate them with the required characteristics. For example, if the emulation should create a link with 100 Mb/s capacity and we can measure its practical maximum bandwidth, it is trivial to check if the link requirements are realised or not.

3.3 The Algorithm in Detail

The presented algorithm that targets the motives described in Section 3.1, namely, to provide a cost-efficient and reliable resource allocation algorithm for clustered network emulator solutions that can run creditable experiments on emulated networks in the cloud. Obviously, there is a slight trade-off between cost-efficiency and reliability. In order to provide creditable measurement results which are important to suit for academic usage as well as for production, the reliability must be prioritised over cost-efficiency. Accordingly, the algorithm is designed to provide a reliable and also a cost-efficient resource allocation.

As the figure of algorithm's naive version (Figure 4), shows, the algorithm's input is the description of the network to emulate. This description contains the parameters of the emulated network (topology, link characteristics, resource limitation of emulated hosts, mapping constraints, etc.).

Based on the input, the next step of the algorithm is the discovery of the arbitrary nodes of the cluster and the resources available on them. For the purpose of network emulation, the resources mentioned in Section 3.2 as very important to monitor are the most important ones at resource discovery as well. These resources are namely CPU performance, memory capacity and quality of links.

The results of this discovery is used in the next step of the algorithm. In this step the discovered nodes are put into a list (mentioned as node-list) that is ordered by the predicted network emulation performance of the nodes. The first element of this list is the most powerful node, and the last one is the least powerful

according to a predicting function. This function that predicts the performance of a node is not pre-wired, because, the costs of the resources depend on the tariffs by the cloud provider. Because this function is the one of the key aspects of cost-efficiency, choosing it carefully is required to achieve maximal cost-efficiency.

After the ranking of nodes, the algorithm tries to embed the network onto the first n number of computational nodes of the node-list using a placement algorithm that is either provided by an existing network emulator or built from scratch. This placement algorithm must be able to embed the arbitrary segments onto the arbitrary nodes and interconnect them by virtual links. In parallel with the embedding, the monitoring of operation critic resources which are described in Section 3.2) begins.

When monitoring detects any threat such as there is not enough free processor resource to resume the emulation without any disturbance, then it will interrupt the emulation and this interrupt will be a feedback that will launch a new iteration of network embedding with a bigger resource set (with more computational nodes) than before. Otherwise, the algorithm terminates when the emulated network terminates.

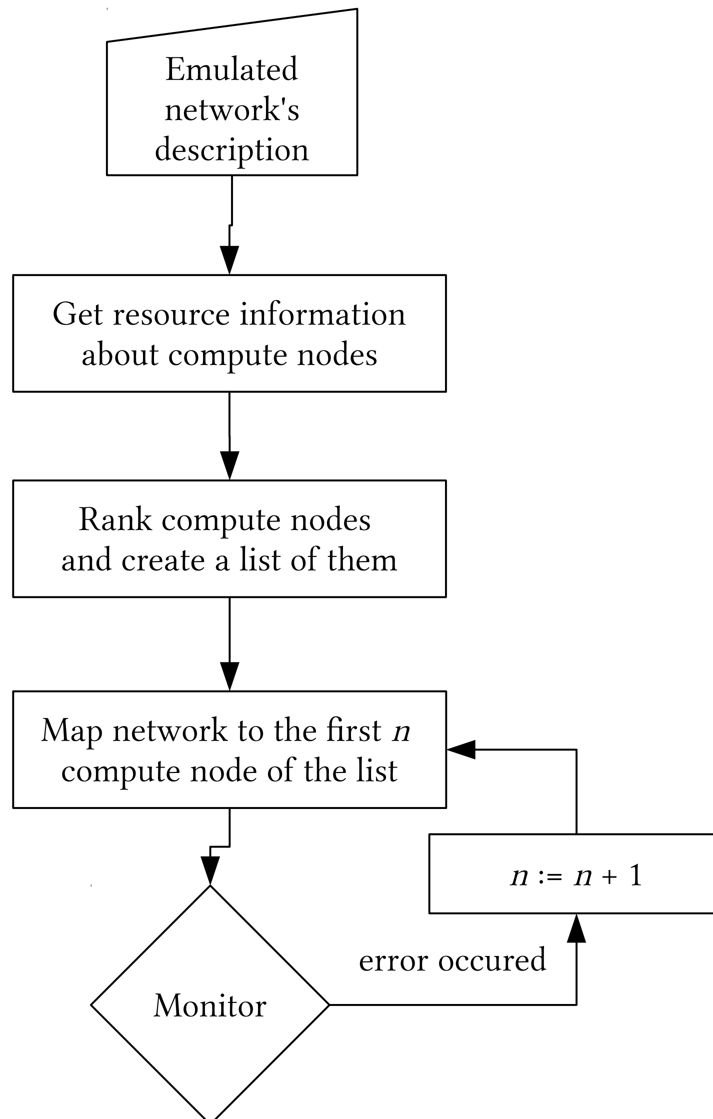

Figure 4. Flowchart representation of the naive algorithm.

Additional remarks for the steps of the algorithm:

1. Criteria (e.g., link bandwidth, host CPU limit) given in the input network description can provide useful information for the next steps of the algorithm.

2. There are diverse methods to discover resources of nodes (examples can be

seen in Section 3.2), but there is the problem of comparing these results between different software (e.g., operating system) or different hardware (e.g., CPU architecture). One solution for this issue might be to run a dedicated CPU intensive test application and measure its completion time on the hosts at resource discovery time.

3. It seems trivial to rank the nodes with all the relevant resource information, however, it is not the case. Because there is a huge set of non-technical aspects such as management and economic considerations: different cloud providers price resources differently. To achieve maximal cost-efficiency, these aspects must be considered and the weights in the comparing function should be set accordingly to the tariffs.

4. The presented algorithm can rely on the placement algorithm of a network emulator when it tries to embed a network. Fundamentally, there is no need for complex algorithms at that stage, the simple ones can be very effective in many real life scenarios. For example, it is typical that the cluster consist of nodes with the very same resources (e.g., same hardware); in this case the bin placer algorithm which places the network elements to emulate evenly on the nodes of the cluster knowing the number of nodes. This algorithm in this use case is incredibly fast, scales well and easy to implement.

   The granularity of the iteration steps during embedding tries, namely, how many hosts should it activate at each iteration, is determining for efficiency. For networks that are as large as they require more then one or a couple of nodes to embed properly, it is pointless to try to embed these networks onto smaller number of nodes than they require. Therefore, these superfluous steps can be omitted, and hereby it increases the efficiency of the algorithm. The number of required hosts for a virtual network to embed can be put into a database to speed up the re-embedding process of the same network on the same nodes.

   As oppose to these optimisation, the naive algorithm which can be seen in Figure 4 uses very simple stepping. It starts with the first element of the

node-list and adds additional elements one by one to the resource set for embedding. The main advantage of this solution is the simple implementation, but, it is far from the optimal solution.

Resource requirements of the emulated network can be queried from the emulated network's description. If it does not define its requirements explicitly, it is still possible to forecast the capabilities of the nodes (how many and what kind of network elements can it run) by heuristics and performance measurements and embed the network accordingly. Due to the possible error of the forecast, there is an extra need for monitoring.

5. There are multiple paradigms to implement monitoring that follows the aspects described in Section 3.2. The cluster nodes involved in network emulation can be monitored in a centralised or in a distributed manner. In case of centralised approach the monitoring actions like measuring and data querying are done by the central entity which also runs the implementation of the algorithm. By the other approach the monitoring is distributed: the nodes measure their resource utilisation separately, then aggregate and forward their measurements. The big advantage of this approach is the even distribution of monitoring overhead over the active nodes of the cluster, however the implementation and installation of distributed monitoring is more complex than central monitoring.

   During the implementation of the algorithm one must take account of the monitoring overhead, and should endeavour to keep this overhead low in order to achieve high cost-efficiency. If the network emulator provides such capability, it is a good solution to rely on its functionality during implementation.

3.4 Proof of Concept

To check the viability of the concept behind the algorithm presented in Section 3 a proof of concept was implemented in Python utilising the cluster support of Mininet. It wraps Mininet transparently, so each Mininet command line argu-

ment provides the same functionality as it would provide in vanilla Mininet except `-cluster`. Due to the heavy collaboration with Mininet, the proof of concept was developed under the code name Micronet referring to its frugal resource utilisation. The installation of Micronet is the same as Mininet's. It runs on one node and requires no extra software.

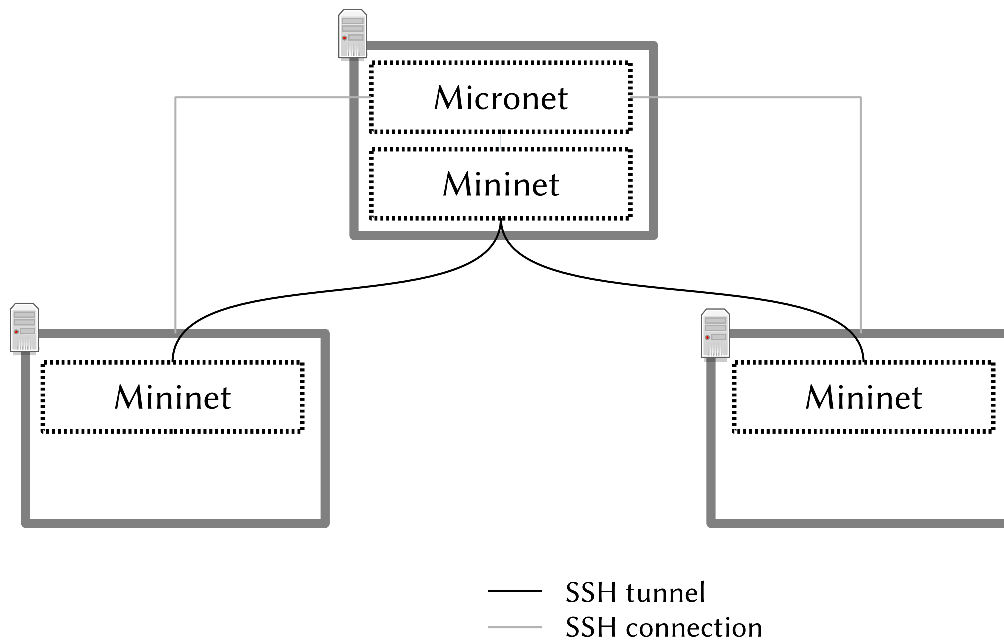

Figure 5. Infrastructure of the Proof of Concept.

The infrastructure of the implementation is simple (see Figure 5). It runs on one node in the cluster (this node will be referred as "head") and accesses other nodes via secure shell connection. During its operation, it collects resource information and manages other nodes via this ssh connection. The resource information are queried from operating system provided statistics, and the network emulation functions are provided by Mininet, so this proof of concept can focus only on the implementation of the trial and error resource allocation algorithm presented in Section 3.3, and its required monitoring functions described in Section 3.2 in details.

During its run the application first parses its command line arguments search-

ing for Mininet's "–cluster" argument to gather the host names of the nodes of the cluster. The program then discovers these hosts based on their names: gets their IP addresses and their available resources which are required for ranking the nodes. Currently, the program can only query the processor performance and memory capacity of the node from the system's statistic files on local host and on remote hosts via ssh. The processor performance is represented in bogomips read from the CPU statistic describing `/proc/cpuinfo`. However, bogomips is not necessarily provides a valid image of the processor's performance, it still provides a constant and easy to access metric about the processor. Therefore, it is good for concept proving purposes. The memory capacity is represented by the sum of the available memory in the system. This information is provided by `/proc/meminfo`. In addition to CPU performance and memory capacity, the quality of the links between the given node and the head node is measured. The two metric used here are round trip time and maximum practical bandwidth. The round trip time is determined as an average result of three ping. The measurement of maximum practical bandwidth is a bit more complicated. Namely, the the measurement is carried out with iperf. The head node is in the role of the server, and the actual node to measure is the client. If the measurable node is the local host, these measurements are omitted, because these measurements are needless due to the all-importance of the local host . This all-importance will be further detailed.

After the resource discovery, the program, strictly following the algorithm, ranks the nodes according to their resources. During this ranking the local host has an all-importance even if it hass the lowest amount of resources in the cluster. This artificial modification of the ranking is required to harvest the low management costs of the local host. Namely, there is no need to build ssh connection to manage and monitor the local host, so the overhead of a monitoring ssh connection can be saved. The ranking algorithm considers two factors. One is the local resource set (CPU and memory) of the node, the other is the quality of the link to the node (bandwidth, rtt).

After node ranking the program tries to embed the given network on the nodes. The proof of concept uses the naive algorithm to embed the given network onto

the nodes of the node-list's ever increasing prefix sub list. So, in first iteration it tries the local host which is the first element of the list, then goes one-by-one to embed the network onto that subset of cluster nodes. Each iteration starts with the initiation of monitoring threads. This threads starts monitoring with a small delay and in one or two seconds periodically check the current CPU and memory utilisation on the active hosts. After starting monitoring, the program fires up the required Mininet network with its parameters. In this step Mininet builds up the cluster using only the nodes from the node-list, places network elements, provides user interface, etc. If there are not enough free resources on the hosts, so starting Mininet fails, a new iteration of the embed probing begins with an additional node added the node list if there are still inactive nodes. If all of the available nodes are used and the cycle can not add utilise new node, the allocation fails. The allocation is also unsuccessful if the monitoring threads find high resource utilisation threats that can harm the validity and the authenticity of the on-going experiments and measurements on the emulated network.

4 Evaluation

4.1 Evaluation setup

The evaluation of the proof of concept took place on a PC with a dual-core 2.6 GHz Intel Core i5-3320M and 16 GB of RAM. The base operating system was Debian 8 (Jessie). The used virtualisation software was Oracle VirtualBox 5.0.6 and the virtual machines were instantiated from official Mininet 2.2.1 images (mininet-2.2.1-150420-ubuntu-14.04-server-amd64) with the default configuration (1 CPU core at full capacity, 1024 MB RAM) using one NAT network.

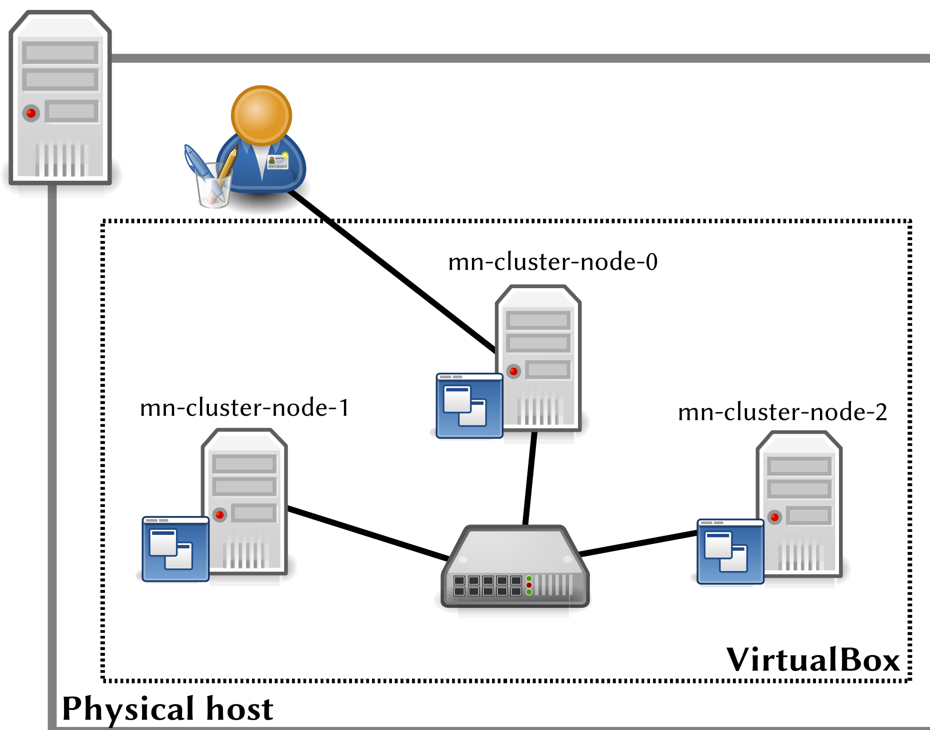

Figure 6. Infrastructure used for evaluating the proof of concept.

The infrastructure of the evaluation can be seen in Figure 6. There was one physical machine on which VirtualBox emulated three almost identical Mininet virtual machines in the the same network. The virtual machines were named as `mn-cluster-{0,1,2}`. These VMs formed the cluster infrastructure for

the evaluation. The configuration of the cluster was done from mn-cluster-node-0 because only that one was accessible from the outside world via its second network interface.

Some preparation was required in order to fire up Mininet Cluster Edition in this setup, however, the official Mininet VM images were used. Each VM was the copy of the first one, therefore, setting arbitrary host names and configuring static routing and host name resolution was the first thing done during installation. After that, some configuration on the ssh daemon was required. Disabling DNS usage was written in the documentation, permitting tunnelling and increasing the maximum number of parallel sessions was mentioned only as a workaround for some issues, however setting these parameters is necessary. The last configuration step was distributing ssh keys over the cluster nodes by the given cluster setup shell script. This script can set the keys temporarily and permanently. Temporal key configuration is useful for ad-hoc clustering which was not the case for the evaluation, therefore, the keys set up was done permanently. Later, the ssh compression was activated and ciphers were changed to blowfish-cbc and arcfour due to their low resource usage in order to increase the bandwidth between nodes.

The proof of concept was copied to the home folder of mininet user on mn-cluster-node-0 via ssh, and was installed by its install script which downloads and sets up the required tools. By issuing these steps, the test bed was ready to use.

4.2 Results

The viability of the concept was justified by two types of measurements. One group of measurements were targeted to confirm that the simple implementation of the algorithm can be so efficient that it can overcome the resource allocation of the de facto network emulator, Mininet. The metric of efficiency was the number of active nodes. A small number of active nodes is important, beside its economically justification, because of the small number of slow and cumbersome node interconnecting links. The other group of measurements was targeted to determine the amount of overhead generated by the required constant monitoring of the approach. Both group of measurements were carried out in the testing setup

described in Section 4.1.

For these comparative measurements tree topology (Figure 1) was used with various depth and fanout parameters. Depth sets the level of the tree and fanout sets the number of branches of a single tree element. Tree topology was used because of the large number of hosts to emulate. Hosts have higher resource requirements than switches, therefore they are more relevant.

First measurement was targeted to demonstrate the frugality of the implementation. This measurement was about embedding a simple network that could even run on a single node, on an increasing number of hosts. As it can be seen in Figure 7, Mininet uses all the available nodes to embed the virtual network no matter of how minimal is the resource set that is required for the network. On the other hand, the implementation of the trial and error algorithm utilises only the required amount of resources.

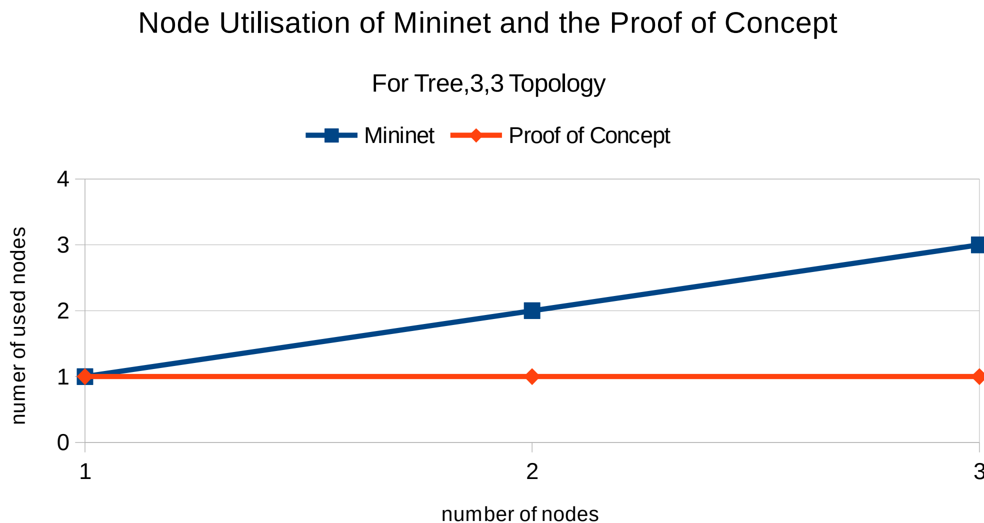

Figure 7. Node Utilisation of Mininet and the Proof of Concept.

In the second measurement scalability was tested. There was a fix number of available nodes in the cluster, but the network to emulate was changing in a sequence of increasing number of hosts. The calculation of host number in a tree topology is simply tree's depth raised to the power of tree's fanout parameter. As

you can see in Figure 8, the implementation scales well as oppose to Mininet which utilises all the available resources in each case.

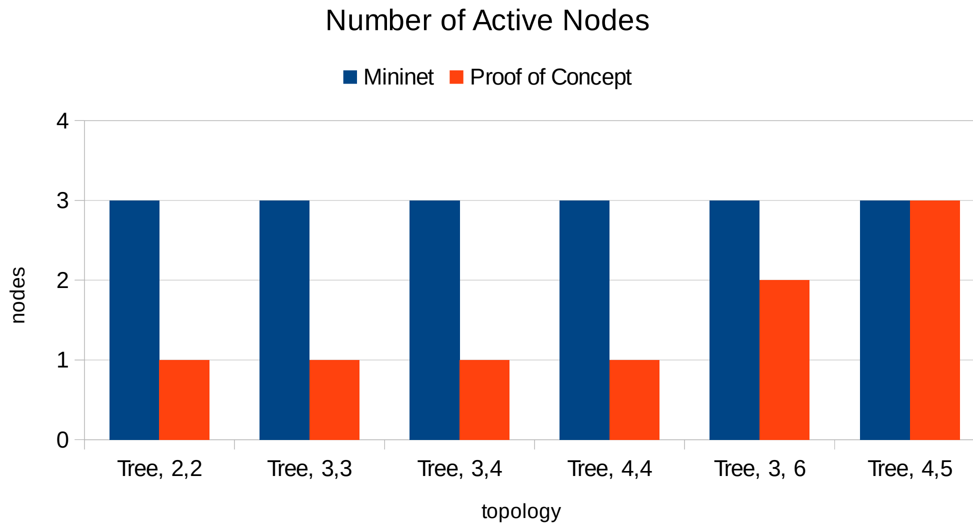

Figure 8. Node Usage of Mininet and the Proof of Concept for Different Tree Topologies.

Based on these measurements, one can say that the algorithm scales well and also economises the available resources well.

For the approximate determination of monitoring overhead, the used CPU cycles, allocated memory and extra network traffic was analysed. As an intuition, the largest amount of CPU and memory overhead mostly came from the ssh connection establishment and termination. Therefore, not just the resource usage of the implementation was monitored, but also the resource usage of the required ssh processes. As Figure 9 shows, the operation of the proof of concept did reached only two percentage of the overall CPU capacity on a single-core 2.6 GHz head node with two additional computational nodes (Figure 6). It is not surprising that the implementation of the algorithm is not CPU intensive – there is no extensive computation in the algorithm. For the CPU bursts secure shell connections are responsible mostly. Fortunately, this load can be lowered by configuring ssh properly with ciphers requiring less resources and disabled compression. Other

method to consider is to use other than SSH remote access technology.

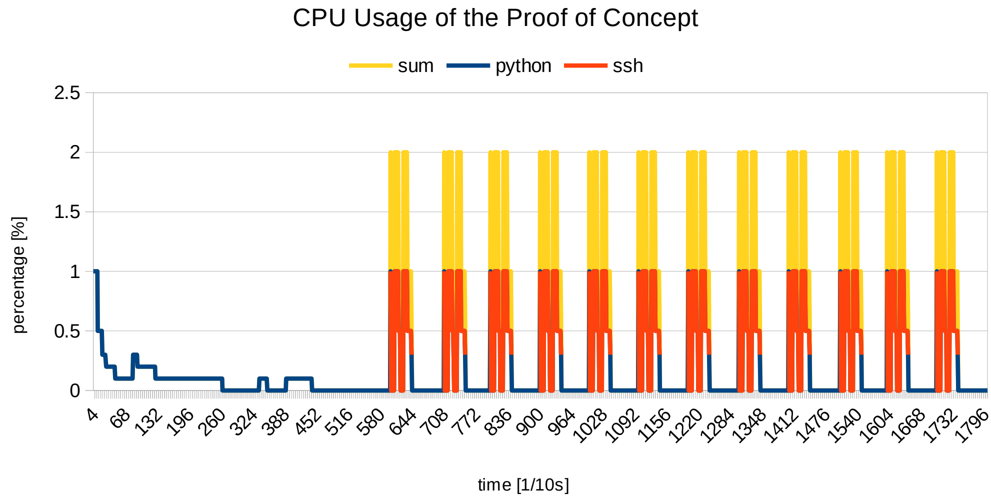

Figure 9. CPU Usage of the Proof of Concept.

The memory usage of the implementation was also low. As Figure 10 plots, the memory usage was 1.2 percentage at maximum on a computer with 1024 megabytes of random access memory and no swap.

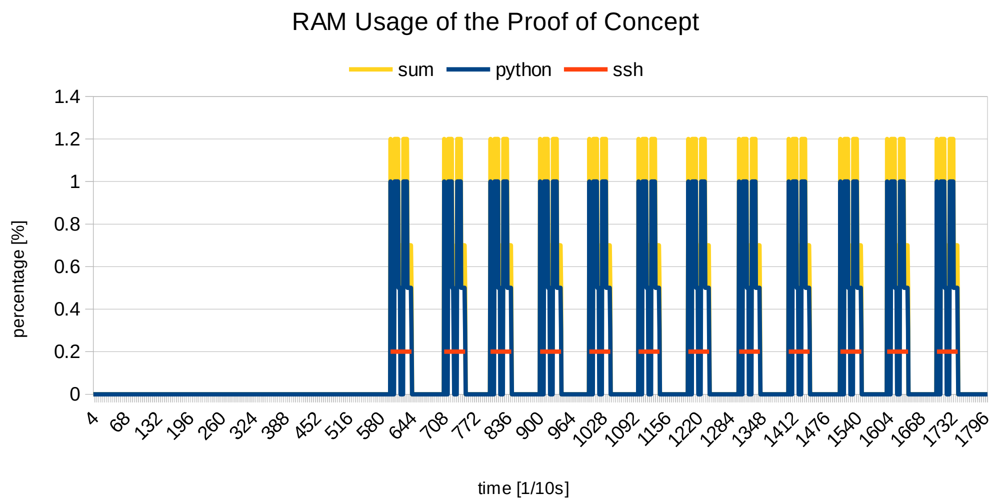

Figure 10. RAM Usage of the Proof of Concept.

The extra network traffic was determined the following way. All traffic was logged during measuring some ssh monitoring cycle, then the relevant ssh traffic was filtered out from the ssh tunnel packets. As these measurements involve sending a quasi fix amount of data due to the fix size of the analysed statistic files, it is possible to logged proportion the measured traffic to the elapsed time and the number of active nodes to determine the extra network overhead's bandwidth. The empiric formula to calculate the bandwidth of the extra network overhead is very simple: measure the traffic of one query from remote node to local node including ssh build up, divide it by the monitoring period time and multiply by the number of active remote nodes. For querying processor utilisation in the evaluation test bed this number was $9878bytes/2seconds * 2 = 9878bytes/sec$ which is practically 9.9 kB/s. For memory monitoring this value was $10378bytes/1second * 2 = 20756bytes/sec$, which is practically 20.8 kB/s. Altogether this two value is 30,7 kB/s which is really low supposing high speed, gigabit links.

To conclude the performance evaluation of the presented algorithm, one can say that the monitoring requires fractional amount of computational resources.

5 Conclusion

5.1 Summary

This document presented existing clustered network emulators in details and stated that these tools are not capable to work in a dynamically changing environment like cloud efficiently. So, to roll back this limitation of them, a possible cost-efficient and reliable resource allocation algorithm was produced targeting experiments involving large scale networks requiring cloud infrastructure and presented.

To better understand this problem, the work also included a proof of concept based on the de facto network emulator, Mininet in order to see the viability of the designed approach. The viability of this trial and error based allocation was proved during its evaluation.

5.2 Future Work

The work presented in this document is not finished, there are many ways to continue.

A straight-forward future work will be to test the approach on real cloud environments such as Amazon EC2.

Also there is still space to improve the proof of concept because it does not utilise all the potential of the presented algorithm currently. This work will be benefitial, because it provides a better understanding how things should be done in a real life scenario and to make further findings (e.g., adaptive iteration) based on these feedback.

6 References

References

[1] Wette, P., Draxler, M., Schwabe, A., Wallaschek, F., Zahraee, M. H., Karl, H. *Wette, Philip, et al. "Maxinet: Distributed emulation of software-defined networks.*, Networking Conference, 2014 IFIP. IEEE, 2014.

[2] *Maxinet website*, `http://maxinet.github.io`, 2015-10-23

[3] Roy, A. R., Bari, M. F., Zhani, M. F., Ahmed, R., Boutaba, R., *Design and management of DOT: A distributed OpenFlow testbed.*, Design and management of DOT: A distributed OpenFlow testbed., Network Operations and Management Symposium (NOMS), 2014 IEEE. IEEE, 2014.

[4] *Mininet website*, `http://mininet.org`, 2015-10-23

[5] Lantz, B., O'Connor, B., *A Mininet-based Virtual Testbed for Distributed SDN Development.*, Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication. ACM, 2015.

[6] Gupta, D., Yocum, K., McNett, M., Snoeren, A. C., Vahdat, A., Voelker, G. M., *To infinity and beyond: time warped network emulation*, Proceedings of the twentieth ACM symposium on Operating systems principles. ACM, 2005.