# MaxiNet: Distributed Emulation of Software-Defined Networks

Philip Wette, Martin Dräxler, Arne Schwabe,
Felix Wallaschek, Mohammad Hassan Zahraee, Holger Karl
University of Paderborn
33098 Paderborn, Germany
Email: firstname.lastname@uni-paderborn.de

***Abstract*—Network emulations are widely used for testing novel network protocols and routing algorithms in realistic scenarios. Up to now, there is no emulation tool that is able to emulate large software-defined data center networks that consist of several thousand nodes.**

**Mininet is the most common tool to emulate Software-Defined Networks of several hundred nodes. We extend Mininet to span an emulated network over several physical machines, making it possible to emulate networks of several thousand nodes on just a handful of physical machines. This enables us to emulate, e.g., large data center networks. To test this approach, we additionally introduce a traffic generator for data center traffic. Since there are no data center traffic traces publicly available we use the results of two recent traffic studies to create synthetic traffic. We show the design and discuss some challenges we had in building our traffic generator.**

**As a showcase for our work we emulated a data center consisting of 3200 hosts on a cluster of only 12 physical machines. We show the resulting workloads and the trade-offs involved.**

## I. Introduction

Today, more and more OpenFlow-enabled data center switches [1] are available on the market but still data center operators do not use them in practice. This is mostly because it is unclear how OpenFlow behaves in scenarios with data center traffic properties and how well existing OpenFlow controllers scale to the size of the data center at hand. Network emulations can help in providing virtual environments with realistic properties to test OpenFlow in data centers before deploying it in practice.

When evaluating new algorithms or protocols for Software-Defined Networks (SDN), emulation using Mininet [2] is the first choice. Mininet uses network namespaces to create separate network contexts for each process running together on one physical machine. With this technique, several OpenFlow-enabled software switches can be run on one machine interconnected by virtual network interfaces. In low-traffic scenarios, Mininet scales to several hundred nodes on state-of-the-art hardware. But when emulating large networks with both high link bandwidths and high traffic volume, the computational complexity of the emulation overwhelms today's computers. In such scenarios it is still possible to successfully emulate the network by using a technique called *time dilation* [3]. Time dilation slows down the emulated time with respect to the walltime by a linear factor (called *dilation factor*) but keeps the speed of peripheral devices constant. By setting the dilation factor to 10, one second of a 10 Gbit link can be emulated by using 10 seconds of a 1 GBit link. Of course, using time dilation, the amount of walltime required for the experiment is multiplied by the dilation factor.

ISBN 978-3-901882-58-6 

We show that the relation between the size of the emulated network and the required dilation factor is *not linear*, leading to unacceptable run times for large network emulations. We suspect this non-linearity is due to the huge amount of required network namespaces, processes and virtual network interfaces, which adds a high amount of overhead to the system and causes the host operating system to work inefficiently. The required dilation factor can be reduced by *distributing the emulation* over multiple physical machines. This is a very time-consuming and error-prone process when done by hand because it has to be decided a) how to partition the virtual network, b) which partition is emulated on which physical machine, c) how to invoke commands at the emulated nodes to keep the experiment synchronized, and d) how to collect the resulting data from the machines for evaluation. In addition, when using such hand-crafted solutions, measurements have to be made to ascertain that the built system works properly (for example, does not distort the latencies between nodes).

This work presents a framework called MaxiNet to use multiple physical machines for large-scale SDN emulations. The whole process of mapping and deploying the network to be emulated onto the physical environment is transparent to the user. Using MaxiNet is like specifying an experiment with Mininet. We evaluated our system through several experiments and found that it properly replicates the properties of a single-machine emulation environment.

We built MaxiNet to allow us to test new routing algorithms for data center networks. This did not only require a scalable emulation environment but also realistic data center traffic. Today, there are no publicly available traffic traces from data centers but recent studies [4], [5] reported several properties of such traffic. Using their findings we built a traffic generator that produces traffic at flow level. Using MaxiNet we were able to emulate a data center network interconnecting 3200 servers using only a dilation factor of 200 on 12 physical machines. MaxiNet and the traffic generator presented in this work are

open source. They can be downloaded from our website[1]. We also provide preconfigured virtual machine images that can be used to test MaxiNet.

The rest of the paper is structured as follows: Section II presents work related to large-scale SDN evaluation. In Section III we present our emulation environment called MaxiNet. Since we aim at evaluating data center networks Section IV presents our data center traffic generator. Section V presents the experience we gained in large scale data center emulation with MaxiNet. Section VI concludes this work.

Throughout the text, with *worker* we refer to physical machines that are used to compute the emulation. The term *node* is used for switches and end hosts that are emulated. The network that is to be emulated (consisting of interconnected nodes) will be denoted as *virtual network*.

## II. Related Work

When evaluating new algorithms or protocols for SDN it usually comes down to the choice between simulation and emulation for a test implementation. For both directions there is already a number of tools available. For simulation there are OMNeT++ [6], NS3 [7], [8] and some commercial tools. NS3 already has a built-in module for OpenFlow and a comparable extension for OMNeT++ [9] also exists.

For our approach we wanted to be able to run native SDN controllers and to include the effects from a native network stack into our evaluation. It is technically possible to implement both with a simulation framework, but from our point of view it is more straightforward to use emulation instead of simulation.

For emulating SDNs there are two possible tools to be considered: Mininet [2], [10] and EstiNet [11]. EstiNet is a commercial tool, which contradicts our idea of building a system that is freely extensible and usable by other researchers. Thus, we focus on Mininet for our implementation.

The option to simulate or emulate large SDNs has been researched in a number of works: In [12] the authors describe how OpenFlow can be run on top of a simulation engine called *S3F*. This approach is limited to one physical machine, thus the performance of this approach is limited. The authors of [13] show how replacing Mininet with their own tool *fs-sdn* can speed up the simulation of SDNs, but again this approach is limited to one physical machine. In [14] an elastic OpenFlow controller is proposed that grows and shrinks with the amount of routing decisions that have to be made. For evaluation the authors built a hand-crafted emulation testbed based on several Mininet instances that were interconnected by GRE tunnels. The authors of [15] claimed they build a distributed version of Mininet for the purpose of malware propagation analysis. However, they lack a description of how their system works and do not show the adequacy of the results obtained with their system.

With respect to our main use case to emulate data center traffic, the work in [16] investigates the distributed simulation of a multi-tier data center, but does not include the simulation of SDN.

To the best of our knowledge, this is the first work to integrate a physically distributed emulation of SDN together with a per-flow generation of data center traffic.

[1] https://www.cs.upb.de/?id=maxinet

## III. MaxiNet: Distributed Emulation

### A. Requirements

When designing MaxiNet we had the following requirements:

- Centralized programming model that is like specifying an emulation with Mininet
- Linear scaling of the virtual network size with the number of physical machines
- Leverage the original Mininet
- Small (physical) network footprint

To achieve these goals we had to solve different problems. The first problem is how to partition the virtual network onto the workers such that a) the physical network does not become a bottleneck and b) the workload is evenly distributed over all workers. Traffic from one partition of the virtual network has to reach all other partitions, requiring forwarding across multiple physical machines. This forwarding process must not introduce a noticeable penalty to the latencies experienced by the nodes. Otherwise the partitioning could influence the outcome of an experiment.

As we want to have a centralized programming model, we need to access nodes across the different workers and these workers have to be synchronized. To achieve this, we decided to run all the control logic of an experiment (the course of when to run which command at which node) on one specialized physical machine called the *frontend*. The frontend partitions and distributes the virtual network onto the workers and keeps a list of which node resides on which worker. This way we can access all nodes through the frontend. The frontend itself can also act as a worker and is manually selected prior to the experiment.

### B. Overview

MaxiNet is an abstraction layer connecting multiple, *unmodified* Mininet instances running on different workers. A centralized API is provided for accessing this cluster of Mininet instances. GRE tunnels are used to interconnect nodes emulated on different workers. MaxiNet works as a front end for Mininet that sets up all Mininet instances, invokes commands at the nodes and sets up the tunnels required for proper connectivity.

Figure 1 shows the schematic view of MaxiNet. A network experiment can use MaxiNet to set up, control and shut down a virtual network by using the MaxiNet API. This API is designed to be very close to the Mininet API to ease using MaxiNet when already familiar with Mininet. The emulation as such happens on a pool of workers. Workers are controlled by MaxiNet using the Mininet API. Communication between MaxiNet and Mininet happens through RPC calls.

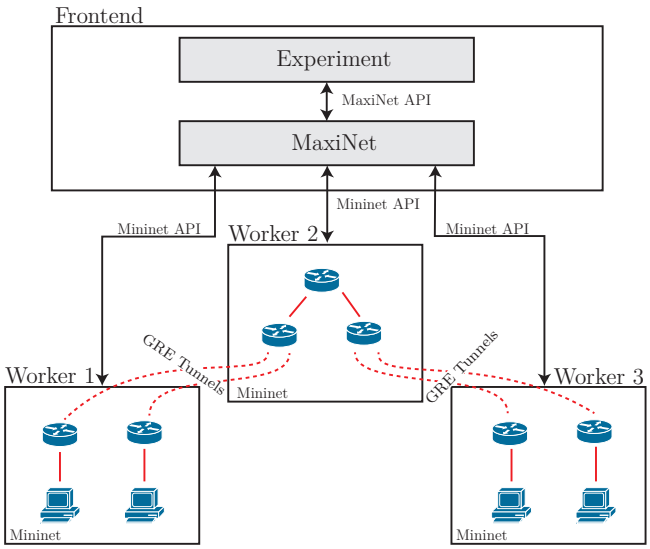


Fig. 1. Schematic view of MaxiNet.

For partitioning a virtual network onto several workers we use the graph partitioning library METIS [17]. For $n$ workers, METIS computes $n$ partitions of near equal weight. The goal of our partitioning process is to confine most of the emulated traffic locally to the workers. The optimization criteria we use for partitioning is minimal edge cut. Edge weights in the partitioning process are proportional to the bandwidth limits specified in the virtual topology. Node weights in the partitioning process are chosen to be proportional to the corresponding node degree in the virtual topology. This makes sense because a node with a higher number of links is likely to forward more traffic, thus causing more load to the worker.

### C. Using MaxiNet

To get the reader familiar with MaxiNet, this section provides a minimal example on how to use MaxiNet. Figure 2 shows the complete Python code (error checking omitted) required to set up and run an experiment on a tree network that is emulated on three different physical machines. First, a `MininetCluster` object is built (line 5). This object holds a list of all $n$ physical hosts' network addresses (here: DNS names pc1, pc2, and pc3) that will be used as workers. The `start` function (line 6) prepares the workers for the emulation by starting a Mininet instance on each machine.

The `Emulation` object (line 8) is the interface to MaxiNet's API. It's creation requires the `MininetCluster` object and a `mininet.topo.Topo` object describing the virtual network topology. In the example we use a tree topology from the Mininet library specified as `TreeTopo(3,2)`. The `addController` function (line 9) is used to specify the OpenFlow controller for the experiment. By calling the `setup` function (line 10), the topology is clustered into $n$ partitions using METIS. After clustering, from each partition a `mininet.topo.Topo` object is built and emulated at a worker. For each edge in the topology that is between nodes

```
1 import sys
2 import maxinet
3 import TreeTopo from mininet.topolib
4
5 cluster = maxinet.MininetCluster("pc1","pc2","pc3")
6 cluster.start()
7
8 emu = maxinet.Emulation(cluster, TreeTopo(3,2))
9 emu.addController("192.168.0.1", "6633")
10 emu.setup()
11
12 print emu.get("h2").cmd("ping -c5 10.0.0.3")
13
14 emu.stop()
15 cluster.stop()
```

Fig. 2. Minimal example experiment using the MaxiNet Python API.

in different partitions a GRE tunnel is set up that directly connects to the interfaces of the nodes.

After invoking the `setup` function of the `Emulation` object the emulation begins. Now, during the run time of the emulation, it is possible to invoke commands at the nodes (line 12). To do so, first the node object is fetched via the `get` function. Afterwards, the `cmd` function can be used to pass a shell command that is executed in the node's environment.

### D. Limitations

The current implementation of MaxiNet has no notion of either the performance of workers or the physical network. This restricts its usage in heterogeneous environments. Load is aimed to be evenly distributed over all workers, so weak machines will become the bottleneck of the emulation. Furthermore, the physical network properties have no influence on the partitioning itself or on the mapping of partitions to workers. Thus, it has to be ensured that the latencies and available bandwidth are the same for every pair of worker nodes. Future work will concentrate on eliminating both these shortcomings.

The experiments in Section V showed that when 12 workers are connected with 1Gbit Ethernet in a star topology the network does not become a bottleneck when emulating a data center with 3200 hosts and a dilation factor of 200.

## IV. Data Center Traffic Generation on Micro-Flow Level

### A. Traffic Characteristics

When it comes to testing new routing algorithms for data centers, most work assumes traffic that is created by a Poison process. But recent studies revealed that the traffic in real data centers is very different from that assumptions [4], [5].

According to [4], the distribution of non-zero entries of a typical data center traffic matrix (TM) is heavy-tailed. They report that for a pair of servers located in the same rack, the probability of communicating in a fixed 10 s period is 11% whereas the probability for out-of-rack communication for any pair of servers is only 0.5%. In addition, a server either talks to the majority of servers in its own rack or to less than one forth of them. The amount of traffic that is exchanged between

server pairs is distributed based on their relationship: Servers in the same rack (non-zero TM entries) either exchange only a small amount or a large amount of data, whereas traffic across racks is either small or medium per server pair.

The authors of [4] found that 80% of the flows in the data center last no longer than 10 s and that only 0.1% of the flows last longer than 200 s. More than half the traffic is in flows shorter than 25 s and every millisecond 100 new flows arrive at the network.

An independent study [5] looked at traffic from 10 different data centers. They showed that across all data centers the flow sizes follow nearly the same distribution. Most of the flows were smaller than 10 KB and 10% of the flows are responsible for more than half of the traffic in the data centers.

We built a versatile traffic generator that is capable of generating traffic with the properties found by the above-mentioned studies. In addition, our traffic generator can be adjusted to create traffic from various distributions. It takes as an input six cumulative distribution functions (CDFs) describing the following properties:

1) Number of in-rack non-zero TM entries per host
2) Number of out-of-rack non-zero TM entries per host
3) In-rack outgoing traffic volume (in bytes) per host
4) Out-of-rack outgoing traffic volume (in bytes) per host
5) Flow size in bytes
6) Inter-arrival time of flows

As additional input, the total number of hosts in the data center as well as the rack size (in hosts) has to be given. We assume each rack in the data center is of equal size.

### B. Generating Traffic Matrices

From the above mentioned inputs, we create a series of traffic matrices (TMs). A single TM describes the amount of traffic exchanged between each server pair in a fixed 10 s period.

To create a TM, for each row $u$ (which corresponds to Server $u$) we first determine the number of *non-zero TM entries*. We do so by sampling a) the number of in-rack and b) the number of out-of-rack communication partners of $u$ using CDFs 1 and 2. The corresponding communication partners are chosen uniformly at random from either a) the servers in the same rack or b) the rest of the servers in the data center. After all communication partners $(u, v)$ (non-zero TM entries) are chosen, these entries are assigned a traffic volume. This volume is sampled from either CDF 3 or CDF 4, depending on the locations of $u$ and $v$. In a subsequent step the traffic volume between any pair of hosts is divided into single flows.

### C. Generating Flows

Data center traffic consists mostly of short-lived flows [4], [5]. Thus, to create data center traffic a number of flows for each non-zero TM entry has to be given. We define a flow to be a series of packets between a source and a destination that are logically belonging together. This could, for example, be one TCP connection. This leads to our definition of a flow as the 4-tuple *(start time, source, destination, size)*.

Generating flows that have to have specific properties (CDFs 5 and 6) for a given traffic matrix is a challenging task. A simple approach would be to go through all non-zero TM pairs $(u, v)$ and sample flows for them according to the CDFs. There are a couple of challenging questions arising from this approach, for example:

*When to stop sampling new flows for $(u, v)$?*
When we stop assigning flows to $(u, v)$ when the sum of flows for $(u, v)$ is larger than specified by the TM, than a lot more traffic would be generated than is specified by the TM. Another way would be to stop sampling flows for $(u, v)$ when the next flow that is to be sampled would exceed the amount stated be the TM. This way, the amount of generated flows would be less than specified by the TM.

*What if for a small TM entry a huge flow size is sampled?*
Resampling the flow size in a situation where a too large flow is sampled for a small TM entry distorts the flow size distribution. And by assigning too large flows the resulting traffic would no longer follow CDFs 3 and 4.

So generating flows for each host pair individually is not practical.

One way to get around these issues is to first create the TM and then a set of "unmapped" flows following CDFs 5 and 6 (where unmapped means the flow is not yet assigned to a source or a target). Afterwards, flows get mapped to source-destination pairs (*s-d pairs*) such that the sum of flow sizes mapped to each s-d pair matches the amount given by the traffic matrix. However, this mapping has to be done very carefully. Since there is no information known about inter-flow dependencies, the mapping must not introduce any artificial patterns to the generated traffic (such a pattern could, for example, be a higher probability to map large flows to node pairs with large TM entries). Thus, the goal is a random assignment of flows to host pairs $(u, v)$ where the amount of traffic given by the flows between $u$ and $v$ is equal to the TM entry $(u, v)$. We call such a mapping an *exact mapping*. Note that it is not guaranteed that an exact mapping exists. Nevertheless, a good mapping strategy gets the amount of flows and the TM entries as close as possible.

To create flows, we first determine the overall required traffic $s_M$ of the TM (as the sum of all entries) and then create a set of unmapped flows matching this overall demand. We denote the sum of all generated flow sizes as $s_F$. Since both $s_M$ and $s_F$ are random variables it will hold that $s_M = s_F \cdot \varepsilon$, $\varepsilon \in \mathcal{R}_0^+$, where $\varepsilon$ is the imbalance factor between the size of the flows and the TM. Of course, $\varepsilon$ should be very close to 1 (meaning there is no imbalance at all), which is why we *resample* the set of unmapped flows with adjusted flow inter-arrival times (CDF 6) as long as $\mid \varepsilon - 1 \mid > 0.01$. This means that the amount of generated flows deviates at most 1% from the traffic specified by the TM. We assume this to be a reasonably small number.

We will now present two different strategies to map the unmapped flows to node pairs. The first one is a purely random

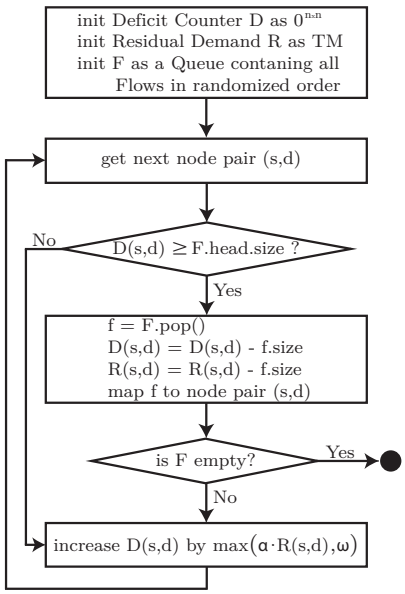


Fig. 3. Selecting s-d pairs for flows using Deficit Round Robin.

process and the second one uses a variation of the *deficit round robin* (DRR) queuing strategy [18]. Afterwards we study the quality of both strategies.

The randomized assignment uses the TM as a probability distribution and, for each generated flow, samples a node pair from this distribution. After a flow has been assigned, the probability distribution at the point of the node pair is lowered by the size of the flow and the next node pair is sampled.

The DRR-inspired strategy can be seen in Figure 3. In the algorithm, $s$ always corresponds to a source, $d$ to a destination and $R$ is the residual traffic as specified by the TM (whenever a flow is assigned to $(s,d)$, $R_{s,d}$ is decreased by the size of the flow). $F$ is a queue that initially contains all flows in a randomized order. $D_{s,d}$ is the deficit counter (due to DRR) of the TM entry $(s,d)$ in the flow assignment. $D_{s,d}$ increases over time and is decreased whenever a flow is assigned to $(s,d)$ by the size of the flow. The algorithm iterates Round Robin over all node pairs and tries to assign the flows queued in $F$. For each flow $f$ the algorithm iterates as long over the node pairs $(s,d)$ as no valid candidate has been found. $(s,d)$ is a valid candidate for flow $f$ if $D_{s,d}$ is larger than or equal to the size of $f$. After a pair $(s,d)$ has been inspected its deficit counter is increased by $\max(\alpha \cdot R_{s,d}, \omega)$. $\alpha$ and $\omega$ control the increase of the deficit counter over time. Ideally, both parameters are chosen to be very small. We found that setting them to values below $\alpha = 0.1$ and $\omega = 100$ has no significant influence on the flow assignment and only increases the run time of the algorithm. Thus, we consider $\alpha = 0.1$ and $\omega = 100$ to be a good choice.

### D. Quality of Flow Assignment

In an optimal flow assignment, each node pair is assigned flows which exactly sum up to the amount of traffic stated by the given TM. To express the difference between the given

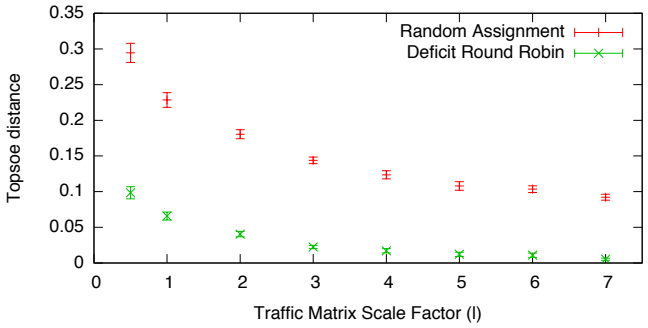


Fig. 4. Topsøe distance of flow assignment methods over different traffic volumes.

TM $M$ and the TM $M'$ produced by the flow assignment we interpret both $M$ and $M'$ as probability distributions of exchanging traffic. Then we express the distance between these two distributions by the relative entropy. The relative entropy is naturally defined as the *Kullbeck-Leibler divergence* (KL), but KL implies that $M'_{i,j} = 0 \Rightarrow M_{i,j} = 0 \; \forall \; (i,j) \in n \times n$ which does not hold in our case. However, the symmetric form of KL, called *Topsøe distance* (see Equation 1) does not have this implication and can be used instead to compute the relative entropy.

$$\text{Topsøe}(M^{(l)}, M^{(l)\prime}) = \sum_{(i,j)} \left( M^{(l)}_{i,j} \ln \frac{2M^{(l)}_{i,j}}{M^{(l)}_{i,j} + M^{(l)\prime}_{i,j}} + M^{(l)\prime}_{i,j} \ln \frac{2M^{(l)\prime}_{i,j}}{M^{(l)}_{i,j} + M^{(l)\prime}_{i,j}} \right) \quad (1)$$

Given a fixed flow size distribution, an increasing communication volume (TM size) will influence the results of the flow assignment methods: If the total traffic volume tends towards infinity, a single flow gets very small compared to a TM entry. In such a scenario it is very easy to find matching flow assignments. We thus look at the quality of the methods under *different* amounts of traffic. To this end, we consider different load levels. A load level is created by linearly scaling CDFs 3 and 4 (which determine the size of the non-zero TM entries) by factor $l$. We denote the corresponding TM with $M^{(l)}$. We then assign flows for $M^{(l)}$ to s-d pairs and calculate the TM $M^{(l)\prime}$ based on that flow assignment. The data center for which we generate traffic consists of 75 racks with 20 servers each. It is the same size that was used for the study [4].

We use $M^{(l)}$ as the ground truth and express the difference between $M^{(l)}$ and $M^{(l)\prime}$ as the relative entropy of both matrices. Figure 4 shows the relative entropy of both the random strategy and the Deficit Round Robin strategy calculated as the *Topsøe distance* (see Equation 1) from 40 matrices of 10 s generated traffic each. It can be seen that for both methods the Topsøe distance decreases with increasing load but for Deficit Round Robin the relative entropy is much lower, thus the method achieves a better flow assignment than the random mapping process.


Fig. 5. One switch pair consisting of 2 switches and 2 hosts connected in a line.

## V. Performance Evaluation of MaxiNet

### A. Scalability

*1) Environment:* The small-scale experiments presented in this section are all run on a cluster of 4 Intel i7 3.3 Ghz quadcore servers with 24 GB ram interconnected by 1 Gbit Ethernet in a star topology. The software switch we used is the Openflow 1.0 userspace reference implementation. We did not use OpenVSwitch because even in its latest version (2.0) OpenVSwitch does not scale well in scenarios where a high amount of switches is emulated at a single host. When emulating a fat tree of depth 7 the throughput of the single switches was highly fluctuating, leading to very bursty traffic patterns. We thus recommend using the userspace reference implementation when aiming at emulating a very high amount of switches, even though the forwarding performance is not as high as for OpenVSwitch.

*2) Single Worker:* To understand the scalability of a single Mininet instance, we run the following experiment: We create pairs of switches with one host each that are interconnected in a line by 10 Gbps links (see Figure 5). Two flows are created (one for each direction) to fully utilize the links. To find out the lowest possible dilation factor we monitored the data rates of the links and lowered the dilation factor as long as all links were fully utilized. The dashed line in Figure 6 plots the number of emulated 10 Gbit links against the required dilation factor (by choosing a smaller factor the processing power of the worker does not suffice to forward all packets). Note that each emulated switch pair corresponds to 30 Gbit. It can be seen that due to the introduced overhead per node the scaling is *not linear*.

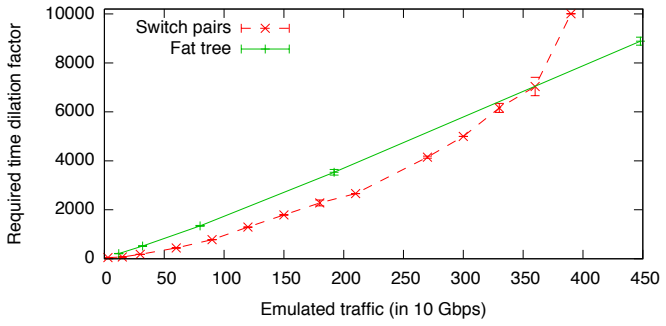


Fig. 6. Required dilation factor when emulating on a single physical machine. Confidence interval for a confidence level of 0.95.

We now look at fat trees with full bisection bandwidth and a data rate of 10 Gbps on the lowest (ToR) level. Each ToR switch is connected to one host for traffic generation. We again aim at fully utilizing every single link while keeping the dilation factor as low as possible. For a fat tree with $n$ ToR switches we create flows between ToR $i$ and $n - i$ to accomplish full utilization of every link in the topology. The solid line in Figure 6 shows the outcome of this experiment: the fat-tree scenario has a nearly linear scaling which is due to the small amount of emulated switches and hosts in comparison to the line scenario. The largest fat tree used in this experiment had a depth of 6 resulting in 128 switches and 64 hosts. The largest switch-pair scenario consisted of 150 pairs, which makes 300 switches and 300 hosts.

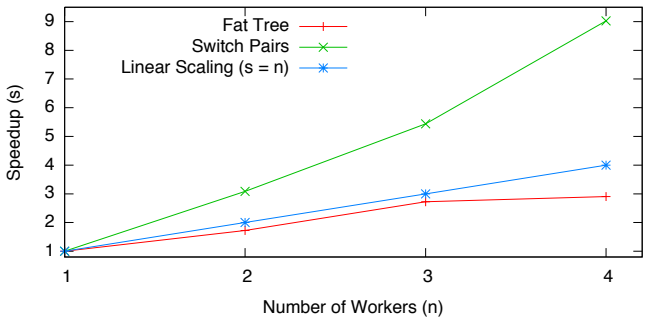


Fig. 7. Speedup gained from an increasing number of workers in the fat-tree and switch-pair scenarios.

*3) Multiple Workers:* We now show the effect of distributing the two scenarios onto multiple workers using MaxiNet. For the experiment we use 120 switch pairs in the switch-pairs scenario. This results in 240 switches, 240 hosts and 360 emulated 10 Gbit links with an aggregated traffic of 3600 Gbps. For the fat-tree scenario we fix the number of leaf nodes to 64, resulting in a fat tree of depth 6. The number of switches is 128 and the number of hosts is 64. The aggregated bandwidth in this scenario is 4480 Gbps and hence off the scale for a single Mininet instance. The two scenarios are chosen to be the best case for the emulation, respectively the worst case because in the switch-pair scenario the virtual network consists of many isolated and thus independent network components and in the fat tree *all traffic* is routed through one single switch (the root of the tree). Since the root can only be emulated at one worker this affects the performance of the whole network and restricts the possible scalability when distributing to a cluster of workers.

Figure 7 plots the speedups over the number of used physical workers. For the switch-pairs scenario, the speedup is better than linear which is due to the lower overhead of smaller virtual networks. When using only one worker, the amount of nodes and virtual links is very high, leading to severe performance penalties (the Linux kernel has to handle 720 virtual interfaces and 360 veth pairs *and* do traffic shaping on them). When distributed over multiple workers the overhead gets lower and no longer hurts performance.

It can be seen from Figure 7 that the effect of adding more workers to the fat-tree scenario does not have a large effect on the speedup. There is basically no gain from adding a fourth worker to the cluster. As already said, this is due to the root

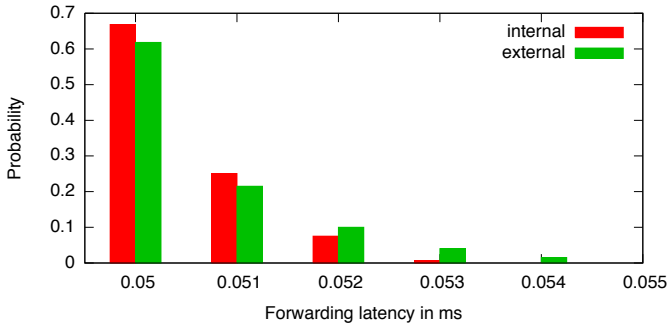


Fig. 8. Distribution of forwarding delays between nodes emulated at the same/different worker nodes.

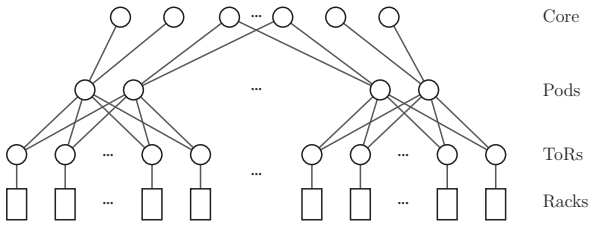


Fig. 9. Sketch of the emulated Clos-like topology.

node of the fat tree. The experiment is designed such that all traffic is routed over the root switch. That switch can only be emulated at one single worker and thus its CPU becomes the bottleneck of the emulation.

*4) Latency distribution:* Since we are using GRE tunnels over a physical network infrastructure to interconnect the different workers, we now take a look at the latencies between hosts emulated at the same physical machine and nodes emulated at different machines. To make latency measurements we use the fat-tree topology described above with 128 leaf nodes and a dilation factor of 5000, emulated at 4 worker nodes. Each link has an emulated delay of 0.05 ms. With the dilation factor this means a latency of 250 ms was configured for each link. We use UDP flows and let them create a utilization of 30% on each link. During the experiment we perform all-to-all latency measurements. The latency histogram between nodes emulated at the same worker (internal) and nodes emulated at different workers (external) are plotted in Figure 8. Both distributions do not differ significantly from each other. This is because the physical network only adds a negligible latency in comparison to the used dilation factor. The same holds for experiments with a lower number of leaf nodes and a lower dilation factor. These results are omitted due to space constraints.

*5) Effects of a distributed emulation:* When distributing an experiment over multiple workers with MaxiNet, the results should be the same as when running the experiment with original Mininet. To show that MaxiNet does not distort results we ran the following experiment on both original Mininet and MaxiNet: We emulated a data center consisting of 300 servers interconnected in a *Clos-like topology*. Figure 9 shows a sketch of this topology. The lowest layer in the topology are racks of

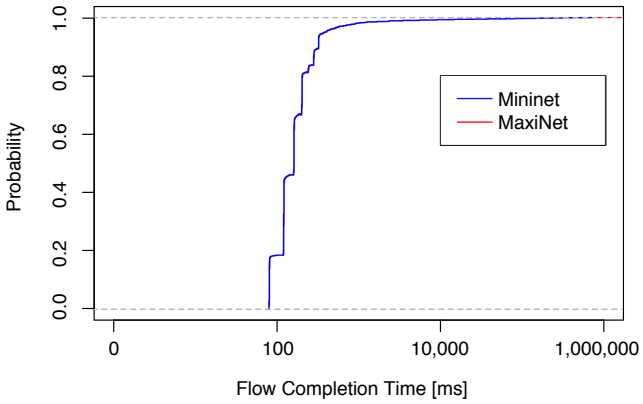


Fig. 10. Distribution of flow completion times between 300 servers emulated at a) one worker (Mininet), and b) four workers (MaxiNet) in a Clos-like network with a dilation factor of 200.

servers. Each rack consists of 20 servers and one top-of-rack (ToR) switch, resulting in 15 racks. Servers are connected with 1GBit links to the ToR switches. *Pods* are formed by grouping three ToR switches and connecting them to two pod switches with 10Gbit links. Each pod switch is connected to two *core switches* with 10Gbit links. In the experiment we used two switches at the core layer. Note that for the comparison, the emulation has to run at a single machine. This is why we decided for such a small scenario.

We emulated 60 seconds of traffic that was generated as described in Section IV. The CDFs we used were extracted from the papers [4], [5]. The dilation factor was set to 200 which means our experiment completed after 200 minutes. We assumed a forwarding delay of 0.05 ms per switch.

Figure 10 shows the CDFs of the *flow completion times* for a) the experiment emulated with original Mininet and b) emulated with MaxiNet on four workers. Note that the results include the dilation factor of 200. The flow completion times in both scenarios follow the same distribution which means that for this particular scenario there is no difference between the results from MaxiNet and Mininet. We cannot think of a reason why this should not also hold for different network sizes and topologies. Hence, we consider the results of experiments with MaxiNet to be *not* distorted when being distributed over multiple workers.

### B. Large-Scale Data Center Emulation

To learn about the scalability of MaxiNet we choose to emulate a data center consisting of 160 racks employing a *Clos-like topology* as before. Each rack consists of 20 servers and one ToR switch, which makes 3600 servers overall. Servers are connected by 1Gbit links to ToR switches. Pods consist of *eight* ToR switches and are connected to two pod switches with 10Gbit links. Pod switches are connected to two *core switches* with 10Gbit links. The core layer in our topology consists of *seven* switches, which makes 207 switches overall. We assume a forwarding delay of 0.05 ms per switch.

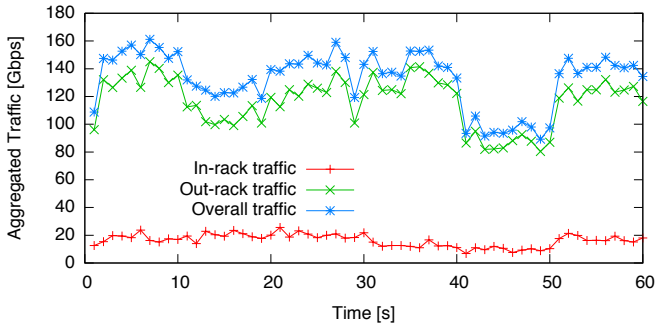


Fig. 11. Aggregate traffic demand used in the large-scale experiment.

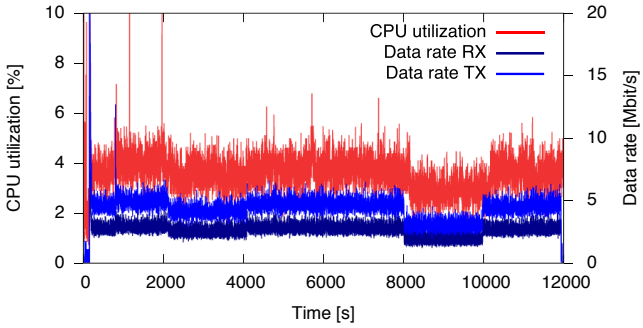


Fig. 12. CPU utilization and network usage of the OpenFlow controller over the course of the simulation.

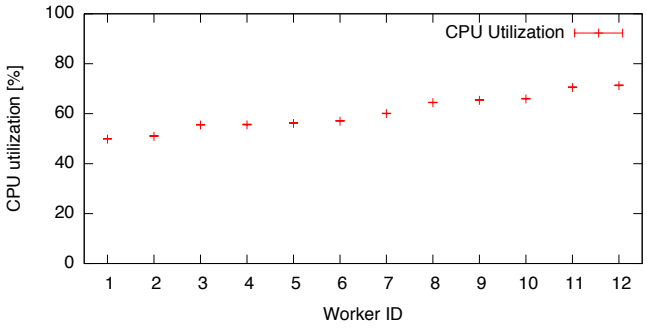


Fig. 13. Average CPU usage of the 12 workers during the large-scale experiment. Confidence intervals with confidence coefficient of 0.95 given.

We emulated 60 seconds of traffic that was generated as described in Section IV and used a dilation factor of 200 which means the emulation completed after 200 minutes. The CDFs were extracted from the papers [4], [5] as before. Figure 11 shows the aggregated traffic demand for both traffic that stays within one rack and traffic that is destined to servers located in other racks. Due to our traffic generation process a new TM is generated every 10 s. It can be seen that the TM for the 40 s – 50 s period contains significantly less traffic than the TMs for other periods, which mimics a low-load phase in the data center. We did not force that to happen; it is just a variation in the random process.

The emulation took place on 12 physical worker nodes that were equipped with Intel Xeon E5506 CPUs running at 2.16 GHz, 12 Gbytes of RAM and 1GBit network interfaces connected to a Cisco Catalyst 2960G-24TC-L Switch. For routing, we implemented equal cost multipath (ECMP) routing based on the Beacon controller platform [19]. As the controller was placed out-of-band and did not use any kind of time dilation, the routing decisions of the single controller were fast enough for the data center network. In addition, the latency between the controller and the emulated switches was not artificially increased. This means that in relation to all the other latencies in the emulated network, the controller decisions were almost immediately present at the switches and did not add any noticeable delay to the flows. Please note that for a real data center (without using time dilation) an ECMP implementation based on only one centralized controller would not keep up with the high flow arrival rates. Figure 12 shows the system utilization of the OpenFlow controller we used for the experiment (which was also run at a 2.16 GHz Intel Xeon E5506). The CPU utilization of the controller is around 4% on average and the data rate required to install rules at the switches is around 5 Mbit/s. When using no time dilation these values would rise to 800% CPU utilization and 1Gbit/s data rate on average. This means for a data center of this size it would require a distributed OpenFlow controller with more than 8 physical machines to run our ECMP implementation. Even assuming a perfect linear scaling of the distributed OpenFlow controller.

Figure 13 shows the CPU utilization of the 12 worker nodes during the experiment. It can be seen that the load is distributed evenly over the worker nodes and that load per worker is very stable. No worker was running at its capacity which means the experiment was not affected by hardware limitations. Figure 14 plots the corresponding network usage over the course of the experiment. It can be seen that the network also does not run at its capacity and that data rates are *not* subject to heavy fluctuation. The later is due to the ECMP routing algorithm and the type of traffic which results in an even distribution of load over the emulated switches. Please note that Figure 14 only plots the portion of traffic that used the physical network and does not include traffic that was exchanged on links connecting nodes emulated at the same worker. This is why there is no strong correlation between Figure 13 and Figure 14.

From the data it can be seen that 12 physical worker nodes are sufficient to emulate a data center consisting of 3600 servers interconnected in a Clos-like topology. The required physical network footprint is very low and the CPU utilization of each worker in this experiment is below 80%. When emulating larger networks, however, either more workers have to be used or a larger dilation factor has to be chosen. Otherwise there are not enough free CPU cycles to compensate for peaks in the CPU usage, which otherwise will bias the outcome of the experiment.

### C. Lessons Learned

To ease the setup of the large-scale experiment we first used MaxiNet in virtual machines (VMs) where each physical

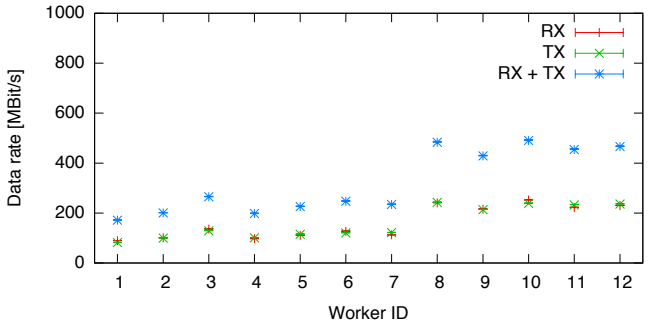


Fig. 14. Average network usage of the 12 workers during the large-scale experiment. Confidence intervals with confidence coefficient of 0.95 given.

worker hosted one VM. We used Linux KVM as the virtualization environment. KVMs network virtualization module, however, is restricted to only one RX/TX queue per network interface. This led to an uneven CPU utilization inside the VMs and thus limited the scalability of MaxiNet. We assume that the same effect will also occur when using physical machines with network interfaces limited to one RX/TX queue. We thus recommend to use network interfaces with multiple RX/TX queues to reduce the required dilation factor.

Choosing the right dilation factor is crucial for the emulation. When choosing a too high factor the emulation takes unnecessarily long to complete, and by choosing a too low factor the results of the emulation are distorted by limitations of the physical network or by the limited processing power of the workers. Unfortunately, the required dilation factor strongly depends on the used virtual topology, the amount of traffic and the software running at the emulated hosts. This makes it hard to give a general statement about the required dilation factor. To find out the smallest possible dilation factor we started with a high dilation factor and decreased it subsequently as long as the result of our experiment did not change.

## VI. Conclusion

With MaxiNet it is possible to emulate data center topologies at scale in a reasonable amount of time using a cluster of physical machines for the computations. This, together with our traffic generator, opens the door for realistic evaluation of novel data center routing protocols through emulation that can be used for rapid prototyping evaluations. We could show that the physical resources required for the emulation of a mid-sized data center are very low even when using acceptable time dilation factors. Our whole emulation environment consisted of old Intel Xeon processors. It can be assumed that by using state-of-the-art hardware even larger data centers can be emulated without using more physical resources or increasing the dilation factor.

## Acknowledgments

This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre “On-The-Fly Computing” (SFB 901).

The research leading to these results has received funding from the European Union’s Seventh Framework Programme (FP7/2007-2013) under grant agreement n° 318115.

## References

[1] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2):69–74, 2008.

[2] Bob Lantz, Brandon Heller, and Nick McKeown. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 19:1–19:6, New York, NY, USA, 2010. ACM.

[3] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C Snoeren, Amin Vahdat, and Geoffrey M Voelker. To infinity and beyond: time warped network emulation. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2. ACM, 2005.

[4] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. The nature of data center traffic: measurements & analysis. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, IMC ’09, pages 202–208, New York, NY, USA, 2009. ACM.

[5] Theophilus Benson, Aditya Akella, and David A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, IMC ’10, pages 267–280, New York, NY, USA, 2010. ACM.

[6] A. Varga. OMNeT++ Discrete Event Simulation System. http://www.omnetpp.org/.

[7] Thomas R Henderson, Mathieu Lacage, George F Riley, C Dowell, and JB Kopena. Network simulations with the ns-3 simulator. In *ACM SIGCOMM demo*, 2008.

[8] S. Guruprasad, R. Ricci, and J. Lepreau. Integrated network experimentation using simulation and emulation. In *Testbeds and Research Infrastructures for the Development of Networks and Communities, 2005. Tridentcom 2005. First International Conference on*, 2005.

[9] Dominik Klein and Michael Jarschel. An openflow extension for the omnet++ inet framework. In *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*, 2013.

[10] Nikhil Handigol, Brandon Heller, Vimalkumar Jeyakumar, Bob Lantz, and Nick McKeown. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, 2012.

[11] Shie-Yuan Wang, Chih-Liang Chou, and Chun-Ming Yang. Estinet openflow network simulator and emulator. *Communications Magazine, IEEE*, 51(9):110–117, 2013.

[12] Dong Jin and David M. Nicol. Parallel simulation of software defined networks. In *Proceedings of the 2013 ACM SIGSIM conference on Principles of advanced discrete simulation*, 2013.

[13] Mukta Gupta, Joel Sommers, and Paul Barford. Fast, accurate simulation for sdn prototyping. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, 2013.

[14] Advait Dixit, Fang Hao, Sarit Mukherjee, TV Lakshman, and Ramana Kompella. Towards an elastic distributed sdn controller. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 7–12. ACM, 2013.

[15] Vitaly Antonenko and Ruslan Smelyanskiy. Global network modelling based on mininet approach. In *Proceedings of the Second ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking*, HotSDN ’13, pages 145–146, New York, NY, USA, 2013. ACM.

[16] Seung-Hwan Lim, B. Sharma, Gunwoo Nam, Eun Kyoung Kim, and C.R. Das. Mdcsim: A multi-tier data center simulation, platform. In *Cluster Computing and Workshops, 2009. CLUSTER ’09. IEEE International Conference on*, 2009.

[17] George Karypis and Vipin Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, 1995.

[18] Madhavapeddi Shreedhar and George Varghese. Efficient fair queueing using deficit round robin. *ACM SIGCOMM Computer Communication Review*, 25(4):231–242, 1995.

[19] David Erickson. The Beacon OpenFlow Controller. In *HotSDN*. ACM, 2013.