Computer Communications 000 (2015) 1–14

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

DCT^{2}Gen: A traffic generator for data centers

Philip Wette*, Holger Karl

Universität Paderborn, Department of Computer Science, Warburger Straße 100, 33098 Paderborn, Germany

ARTICLE INFO

Article history:
Received 27 May 2015
Revised 29 November 2015
Accepted 5 December 2015
Available online xxx

Keywords:
Traffic generator
Data center
TCP schedule

ABSTRACT

Only little is publicly known about traffic in non-educational data centers. Recent studies made some knowledge available, which gives us the opportunity to create more realistic traffic models for data-center research. We used this knowledge to create the first publicly available traffic generator that produces realistic traffic between hosts in data centers of arbitrary size.

We characterize traffic by using six probability distribution functions and concentrate on the generation of traffic on flow-level. The distribution functions are easily exchangeable to enable using up-to-date traffic characteristics whenever new data is available from publications or own experiments. Moreover, in data centers, traffic between hosts in the same rack and hosts in different racks have different properties. We model this phenomenon, making our generated traffic very realistic. We carefully evaluated our approach and conclude that it reproduces these characteristics with accuracy.

1. Introduction

Traffic traces from data-center networks are very rare. This leads to problems when evaluating new networking ideas for data centers because it is not possible to find proper input. We propose a method to generate realistic traffic for arbitrarily sized data centers from only a set of statistical properties of data-center traffic. This enables us to generate traffic for networks where only limited information is available.

Recent studies [1,2] investigated traffic patterns in today's data centers on flow level. They gave a detailed statistical description for both the traffic matrices and the flows present on Layer 2 in data centers. These studies were the first to give a detailed insight into the communication patterns of commercial data centers and reported that different parts of the traffic matrix have different statistical properties. This is due to software tuned for running in data centers (like Hadoop [3]). Such software tries to keep as much traffic in the same rack as possible to achieve a higher throughput and lower latency. We call this property *rack-awareness*.

This paper proposes the Data Center TCP Traffic Generator (DCT^{2}Gen) which takes a set of Layer 2 traffic descriptions and uses them to generate Layer 4 traffic for data centers. When the generated Layer 4 traffic is transported using TCP, it results in Layer 2 traffic complying with the given descriptions. With the Layer 4 traffic at hand, TCP dynamics can be included into the evaluation of novel networking ideas for data centers with pre-described properties of Layer 2 traffic without having to know the exact applications running in the data center. This allows using realistic TCP traffic patterns in experiments conducted at testbeds or network emulations where real network stacks are used. Our generator is highly realistic; e.g. it reflects rack-awareness of typical data-center applications, which enables highly realistic evaluation of novel data-center ideas.

The work flow required to generate artificial TCP traffic and to prove its validity is depicted in Fig. 1. First, Layer 2 traces from the targeted data-center are collected (1). Then, these traces are analyzed to obtain a set of probability distributions describing the traffic (2). These distributions include the number of communication partners per host, the flow sizes, the sizes of traffic matrix entries, and others. From the observed Layer 2 traffic distributions (2) we infer the underlying Layer 4 traffic distributions (3). Using these Layer 4 distributions, we generate a Layer 4 traffic schedule (4). This schedule describes for each host when to send how much *payload* to which other host in the data center. We claim that by executing our calculated schedule, the resulting traffic on Layer 2 (5) has the same stochastic properties as the original Layer 2 traffic traces (1). To prove that, we are using a network emulation to execute the computed Layer 4 schedule. We capture the resulting traffic at Layer 2 (5) from the emulation and analyze its statistical properties (6) to show that these are the same for both the original Layer 2 traces (1) and the generated traces (5).

To compute the Layer 4 traffic schedule, we do not even need to know the exact Layer 2 traces (1). As we only work on statistical traffic properties, it is sufficient to know the Layer 2 traffic distributions, whenceever they come (e.g., easily estimated from available traces

* Corresponding author. Tel.: +49 5251 60 1716.
E-mail address: wette@mail.upb.de, philip.wette@uni-paderborn.de (P. Wette).

http://dx.doi.org/10.1016/j.comcom.2015.12.001
0140-3664/

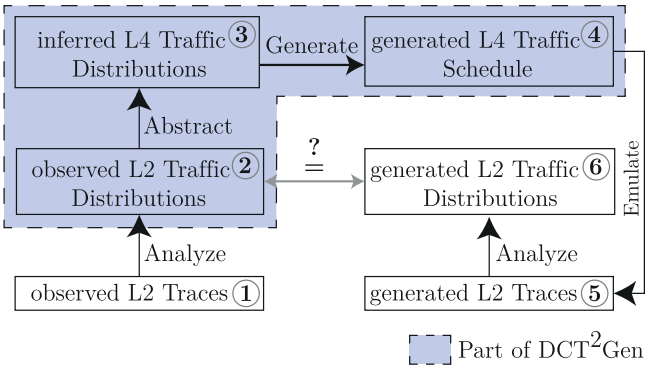

Fig. 1. High-level overview of DCT²Gens work flow.

or observed online from monitoring tools). Also, even if for a data center it is possible to directly obtain Layer 4 traffic distributions (3), DCT²Gen serves as a useful tool to generate Layer 4 schedules (4) from this data. Because even then, it is still necessary to generate traffic matrices from the data and create TCP flows complying with the given distributions.

Finding a Layer 4 traffic schedule (4) is a challenging task because of the bidirectional nature of TCP. TCP is the most common Layer 4 protocol. For this work, we assume that all Layer 4 traffic is transported using TCP and that all TCP connections are *non-interactive* (i.e., payload is only transported into one direction). We have to make this assumption because there is no information about the relationship between the amount of sent and received bytes in interactive TCP connections from data centers available to the public. We want to emphasize that this assumption holds for a lot of applications running in the data center because most of the communication created by data-center applications (such as big-data applications) is due to backlogged data transfers that happen when reading and writing files to a distributed file system.

In TCP, each flow transferring payload between a source s and a destination d also creates a flow of acknowledgments (ACKs) from d to s. The size of this ACK flow is roughly proportional to the size of transferred payload. Thus, half of all flows in the schedule cannot be scheduled arbitrarily. The properties of these flows depend on the other half of flows. This poses a lot of interesting problems that we solved when creating our traffic generator.

DCT²Gen is open source and available for download from our website[1]. We supply all necessary inputs required to generate a traffic schedule complying with the distributions reported in [1,2]. We emphasize that DCT²Gen is independent of these two studies and thus can keep up with the ever changing properties of data-center traffic. Whenever new studies about data-center traffic are published, their results can be used with DCT²Gen, too. To do so, solely the probability distributions (which are given as step functions and are part of the input) have to be replaced. Although we explain, in some parts of this paper, some of our design choices with the behavior of a Map-Reduce-style workload (backlogged data transmissions, almost all TCP connections non-interactive), our assumptions on the traffic in data centers do not depend on Map Reduce and are also valid for traffic produced by other data-center applications because applications in the data center mostly process data which first has to be fetched over the network. And this operation results in backlogged and mostly non-interactive TCP connections.

The rest of this paper is structured as follows. In Section 2 we give a short overview of the landscape of traffic generators. Section 3 discusses traffic properties of data-center networks. These properties have to be replicated by our traffic generator whose architecture is presented in Section 4. Section 5 deals with one of the main challenges of this work: A method is described to find the distribution of the sizes of Layer 4 traffic matrix entries from the distribution of the sizes of Layer 2 traffic matrix entries. Section 6 describes the process of traffic matrix generation. Section 7 explains how to use these traffic matrices to create a schedule of Layer 4 traffic. In Section 8 we evaluate our traffic generator and conclude this paper in Section 9.

[1] https://www.cs.upb.de/?id=dct2gen

2. Related work

Past research has created a large number of different traffic generators, all with different aims and techniques. From our point of view, there are four key characteristics of available traffic generators:

- Flow-level vs. packet-level
- Traffic on one link only vs. traffic on a whole network
- Automatic vs. manual configuration
- Topology awareness vs. non-topology awareness

We give a short overview of each characteristic and afterwards use them to categorize existing traffic generators.

2.1. Flow-level vs. packet-level generators

There are traffic generators [4–6] that output traffic on packet level, formatted due to certain communication protocols. The mix of these packets follows certain rules and probability distributions that are configurable beforehand. However, these traffic generators do not usually implement flows, i.e. packets that logically belong together and that share certain properties like source and destination addresses. A traffic generator that is flow-aware [7–9] always generates packets organized in flows. Flow generation is done such that the flows meet certain statistical properties.

2.2. Traffic on one link only vs. traffic on a whole network

The majority of existing traffic generators [4–6] concentrates on generating traffic originating from one interface only. For performance evaluation of whole network topologies it is required to know the packet stream that is created by each single device in the network. As typically these streams are correlated, it is not sufficient to generate traffic for each interface separately but a traffic generator that creates correlated traffic for a whole network [10] is required.

2.3. Automatic vs. manual configuration

Network traffic has various properties depending on the type of the network. To specify desired traffic properties, traffic generators can be parameterized by hand [11], automatically by feeding *traffic traces* from a real network whose traffic has to be mimicked [9,12,13], or by a combination of both [9,13]. For the automatic case either algorithms are used to extract parameters from the given traffic or the given traffic itself is part of the generated traffic. In that case it is often used as background traffic that is superimposed by special traffic that has properties based on the desired test case.

2.4. Topology awareness vs. non topology awareness

Traffic generators can be *topology-aware*. In that case, the topology of the target network influences the traffic patterns produced by the generator. In the context of data-center networks, the traffic matrices are typically dense in intra-rack areas and coarse in inter-rack areas. Thus, a traffic generator for data center networks has to account for the placement of servers in racks.

Please cite this article as: P. Wette, H. Karl, DCT²Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

2.5. Existing traffic generators

Harpoon [9] is an open-source traffic generator that creates traffic at flow level. It creates correlated traffic between multiple endpoints and automatically derives traffic properties from supplied packet traces. Harpoon is able to generate both TCP and UDP traffic. The general concept of Harpoon is a hierarchical traffic model. Traffic between any pair of endpoints is exchanged in sessions where each session consists of multiple file transfers between that pair of hosts. Sessions can either be TCP or UDP. Harpoon can be parametrized in terms of inter-arrival and holding times for sessions, flow-sizes, and the ratio between UDP and TCP. These parameters are automatically derived from supplied packet traces. As Harpoon is not topology-aware it cannot be used to replicate the special properties of data-center traffic.

Ref. [13] proposes a flow-level traffic generator for networks. It uses a learning algorithm that automatically extracts properties from packet traces. That work focuses on generation of traffic from different applications each with different communication patterns. To this end, *Traffic Dispersion Graphs* are used to model the communication structure of applications. The generator reproduces these communication structures accurately but is less accurate in modeling the properties of flows. In addition, this traffic generator does not capture any structural properties of the traffic matrix.

The Internet Traffic Generator [4] and its distributed variant D-ITG [5] focus on traffic generation on packet-level. Both generate a packet stream that can be configured in terms of the inter-departure time and the packet size. A similar traffic generator is presented in [8]. It generates traffic on flow level for a single internet backbone link.

Swing [7] is a closed-loop traffic generator that uses a very simple model for generating traffic on packet level. Swing aims at reproducing the packet inter-arrival rate and its development over time. Packets are logically organized in flows. However, Swing only generates a packet trace for a single link.

Up to now, there exists no traffic generator that computes a schedule of TCP payload transmissions that can be used to produce Layer 2 traffic that complies with statistical traffic properties given beforehand. DCT^{2}Gen is the first generator to compute such a schedule.

3. Traffic properties

To describe and generate traffic, DCT^{2}Gen uses several stochastic traffic properties. Some of these properties are observed from L2 traces that are given as input, some of them are properties of inferred Layer 4 traffic. The traffic description hinges on how flows behave inside the network.

Throughout the paper, the term *flow* describes a series of packets on Layer 2 between the same source and destination that logically belong together. We distinguish two types of flows. A *payload flow* is a flow which transports payload from source to destination – looking from Layer 4, it transports TCP data packets. Since we assume non-interactive TCP traffic, a payload flow does not include any acknowledgments. Acknowledgments are sent in separate *ACK flows*, which only include TCP ACK segments but no data. In consequence, each TCP connection results in two flows. The structure of these flows is captured by *traffic matrices* described in the following.

3.1. Traffic matrices

A traffic matrix (TM) describes the *amount of data* in bytes (not number of flows) that is transferred between a set of end hosts in a fixed time interval, capturing the pattern of flows in such a time interval. The entry (i, j) of a TM tells how much data is sent from server i to server j.

We distinguish several types of traffic matrices. The primary one is the traffic matrix describing the *observed*, actual traffic on Layer 2; we denote this matrix as $\mathrm{TM}^{(\mathrm{obs})}$. The next matrix corresponds to the *generated* traffic on Layer 2 that is a result of DCT^{2}Gen (compare Box 5 in Fig. 1). Generated L2 traffic is described by the traffic matrix $\mathrm{TM}^{(\mathrm{gen})}$.

Layer 2 traffic is juxtaposed to Layer 4 traffic. Layer 4 traffic is usually not observed (or, at least, not reported in publications) but only generated. We have to distinguish between the *payload* traffic matrix describing the actual data flows and the *acknowledgement* traffic matrix for the flows containing only acknowledgement packets; they are called $\mathrm{TM}^{(\mathrm{PL})}$ and $\mathrm{TM}^{(\mathrm{ACK})}$, respectively. Since the payload flows from i to j give raise to the acknowledgement flows from j to i, these two matrices are interrelated:

$$\mathrm{TM}^{(\mathrm{ACK})}(i, j) = \beta \cdot \mathrm{TM}^{(\mathrm{PL})}(j, i)$$

for some value β to be discussed in Section 6.

Moreover, a Layer 2 traffic matrix is the sum of a Layer 4 payload TM and a Layer 4 ACK TM plus overhead; Section 5 discusses the overheads involved here.

In addition to the layer, traffic matrices also reflect the traffic structure inside a data center. Because of rack-aware applications, traffic inside a rack has different stochastic properties than that between racks – we reflect these differences by separate stochastic distribution functions for the *intra-rack* and the *inter-rack* parts of a traffic matrix.

For either part, we have to describe first the stochastic distribution of the number of nodes a given node i talks to (either in its own or in any other rack). Second, we need a stochastic distribution to describe the amount of bytes that is transferred from i to j, for each node j that i talks to; the intra- and inter-rack cases will in general have different distribution functions.

In summary, we need stochastic distributions separately for (a) the cases of observed and generated Layer 2 traffic and for payload and ACK traffic on Layer 4, (b) the distinction between intra- and inter-rack traffic, and (c) the description of number of communication partners vs. number of transferred bytes. This results in $4 \cdot 2 \cdot 2 = 16$ distribution functions so far.

3.2. Flow sizes

The *flow size* denotes the number of bytes transported by a flow including all protocol overhead. An entry of a traffic matrix describes how much traffic is exchanged in total between a pair of nodes in a given time but it specifies neither number nor size of the individual flows transporting this traffic. The *flow-size distribution* specifies how likely a flow of a certain size occurs on Layer 2.

We distinguish between the flow-size distribution of payload flows, the flow-size distribution of ACK flows, *and* the flow-size distribution of both payload and ACK flows combined. Fig. 2 depicts the relationship between payload flows, ACK flows, and all flows on Layer 2. Each flow transporting payload from node i to node j implies an ACK flow from node j to node i. The size of the ACK flow depends on the size of the corresponding payload flow. Hence, the payload-size distribution implies a certain ACK-size distribution and the convolution of both distributions is the flow-size distribution on Layer 2.

To summarize, we distinguish between three different flow-size distributions.

- The *payload-size distribution* specifies how likely a flow of a certain size occurs on Layer 2 which results from a payload flow on Layer 4.
- The *ACK-size distribution* specifies how likely a flow of a certain size occurs on Layer 2 which results from an ACK flow on Layer 4.
- The *flow-size distribution* specifies how likely a flow of a certain size occurs on Layer 2.

Looking from Layer 2, we cannot tell if a flow is a payload flow or an ACK flow. In the process of traffic generation, DCT^{2}Gen takes

Please cite this article as: P. Wette, H. Karl, DCT^{2}Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

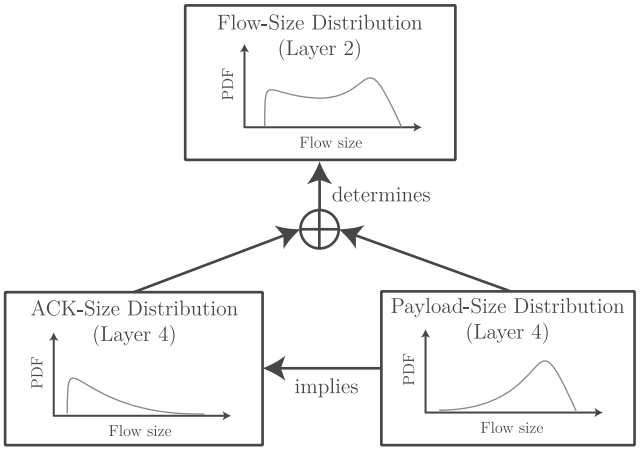

Fig. 2. Relationship between the observable flow-size distribution function at Layer 2 (top), the distribution function of ACK sizes (bottom left) and causal payload sizes (bottom right).

Table 1
Overview of the distribution functions.

Distributions		Observed	Generated	Inferred at Layer 4	
		Layer 2	Layer 2	PL	ACK
Bytes	Intra-rack	B^{obs}_{intra}	B^{gen}_{intra}	B^{PL}_{intra}	B^{ACK}_{intra}
	Inter-rack	B^{obs}_{inter}	B^{gen}_{inter}	B^{PL}_{inter}	B^{ACK}_{inter}
Number	Intra-rack	N^{obs}_{intra}	N^{gen}_{intra}	N^{PL}_{intra}	N^{ACK}_{intra}
	Inter-rack	N^{obs}_{inter}	N^{gen}_{inter}	N^{PL}_{inter}	N^{ACK}_{inter}
Flow size		S^{obs}	S^{gen}	S^{PL}	S^{ACK}
Flow inter-arrival time		IAT^{obs}	IAT^{gen}	IAT^{PL}	IAT^{ACK}

the flow-size distribution of all flows and computes the corresponding payload-size distribution. To be able to infer flow sizes at Layer 4 (from the flow-size distribution) we need to assume that all TCP sessions in the data center are non-interactive. For data centers running mostly Map-Reduce workload this assumption is true for most of the flows.

3.3. Flow inter-arrival time

The flow inter-arrival time distribution describes the time between two subsequent flows arriving at the network. Together with the flow-size distribution, the distribution of the flow inter-arrival time specifies the distribution of the total amount of traffic for a given time interval. This amount of traffic must match the total traffic specified by the corresponding TM. Otherwise, not enough (or too many) flows exist, which means it is not possible to use these flows to create a TM with the desired properties.

3.4. Nomenclature

We shall indicate the distribution functions (*not* the random variables) as follows:

- *N* represents the number of communication partners per node, *B* represents the total bytes exchanged between a pair of nodes, *S* represents the flow size and *IAT* represents flow inter-arrival times.
- The subscript specifies either the intra- or inter-rack case, if needed.
- The superscript specifies the case of observed or generated (on Layer 2) vs. payload or ACK (on Layer 4) traffic.

Table 1 summarizes all the stochastic distribution functions that we use in the remainder of the paper.

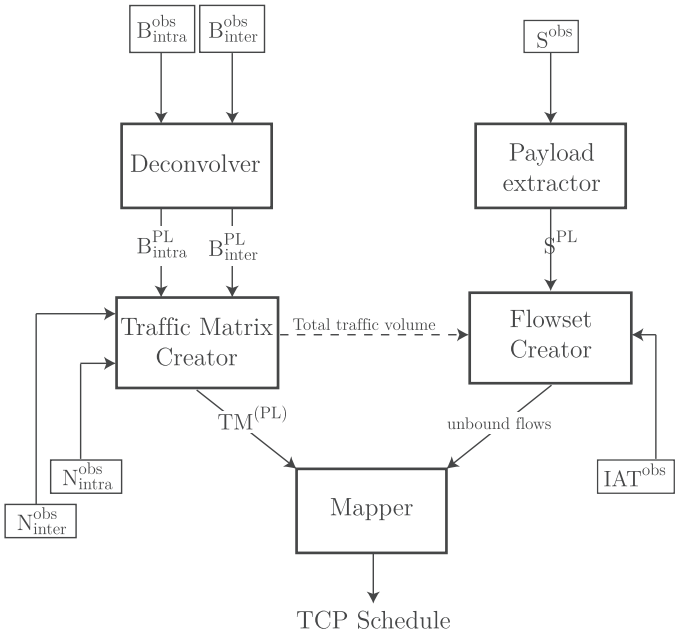

Fig. 3. Architectural overview of DCT²Gen.

4. Architecture of DCT²Gen

We generate a schedule of Layer 4 traffic that specifies at which time how much *payload* has to be transmitted from which source node to which destination node. When transported using TCP, the generated Layer 2 traffic on the network shall have the same properties as the observed Layer 2 traffic. Many possible schedules with this property exist. We aim at finding one of these with the following approach.

First, a $TM^{(PL)}$ is generated. To this end, we need to infer the distribution of payload bytes exchanged by a pair of nodes for the inter-rack case (B^{PL}_{inter}) from the observed distribution of total bytes exchanged by a pair of nodes for the inter-rack case (B^{obs}_{inter}) and the distribution of payload bytes exchanged by a pair of nodes for the intra-rack case (B^{PL}_{intra}) from the observed distribution of total bytes exchanged by a pair of nodes for the intra-rack case (B^{obs}_{intra}). From these distributions, along with the distribution of the number of communication partners per node for the inter-rack case (N^{obs}_{inter}) and the distribution of the number of communication partners per node for the intra-rack case (N^{obs}_{intra}), a $TM^{(PL)}$ can be generated (for details, see Section 6). The next step is to assign flows to all non-zero $TM^{(PL)}$ entries. For this task, we need to infer the distribution of payload-flow sizes (S^{PL}) from the observed distribution of flow sizes on Layer 2 (S^{obs}). The former describes the distribution of the sizes of payload flows that (together with the implied S^{ACK}) generates the given flow-size distribution on Layer 2 (Fig. 2). Using S^{PL}, a set of payload flows is generated which are mapped to the non-zero $TM^{(PL)}$ entries in a subsequent step (Section 7).

At the end of this process we know how many payload bytes to send from which node to which other node in TCP sessions such that it holds that: B^{gen}_{intra} equals B^{obs}_{intra}, B^{gen}_{inter} equals B^{obs}_{inter}, N^{gen}_{intra} equals N^{obs}_{intra}, N^{gen}_{inter} equals N^{obs}_{inter}, S^{gen} equals S^{obs}, IAT^{gen} equals IAT^{obs}

The modular design of our traffic generator can be seen in Fig. 3. It consists of the five different modules *Deconvolver*, *Payload Extractor*, *Traffic Matrix Generator*, *Flowset Creator*, and *Mapper*. This section gives a short description of each single module. In the subsequent sections, complex modules (Deconvolver, Traffic Matrix Generator, Mapper) are explained in detail.

Please cite this article as: P. Wette, H. Karl, DCT²Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

Algorithm 1 Transform S^{obs} to S^{PL}.

1: **Algorithm** InferPayloadSize$\left(Pr^{obs}(\cdot)\right)$:
2: $Pr^{PL}(\cdot) \leftarrow Pr^{obs}(\cdot)$
3: **for each** flow size x in decreasing size **do**
4: $Pr^{PL}(ACK(x)) \leftarrow Pr^{PL}(ACK(x)) - Pr^{PL}(x)$
5: **end for**
6: $Pr^{PL}(\cdot) \leftarrow Pr^{PL}(\cdot) / \sum_x Pr^{PL}(x)$ {normalize $Pr^{PL}(\cdot)$}
7: **return** $Pr^{PL}(\cdot)$

4.1. Deconvolver

The *Deconvolver* takes the observed distribution of total bytes exchanged by a pair of nodes for the intra-rack case (B^{obs}_{intra}) and the observed distribution of total bytes exchanged by a pair of nodes for the inter-rack case (B^{obs}_{inter}) as inputs. From these, it computes the distribution of payload bytes exchanged by a pair of nodes for the intra-rack case (B^{PL}_{intra}) and the distribution of payload bytes exchanged by a pair of nodes for the inter-rack case (B^{PL}_{inter}), which enable us to generate $TM^{(PL)}$. As the name suggests, the Deconvolver uses a deconvolution technique which is explained in detail in Section 5.

4.2. Payload extractor

We need to compute a set of payload flows that, together with the implied ACK flows, generate flows on Layer 2 which comply with S^{obs} (Fig. 2). Flows on Layer 2 are the union of payload flows and ACK flows. As we are only given S^{obs} we need to infer S^{PL} (which itself implies a certain S^{ACK}). S^{PL} is computed in the *Payload Extractor*.

For the Payload Extractor to work, we need to assume that the ratio of payload packets to ACK packets in TCP is fixed at a value r. We substantiate this assumption in Section 5.2 and use r to calculate the ratio of payload bytes to ACK bytes $\beta = \frac{|ACK|}{|PAY|} \cdot \frac{1}{r}$, where |ACK| is the size of an ACK packet and |PAY| is the size of a payload packet. |PAY| is the MTU of the network (which in our case was 1500) plus the size of an Ethernet header (14 Bytes). |ACK| = 66 because TCP Cubic, which is default in Linux, tends to use the TCP Time Stamp option resulting in an ACK packet size of 20 Bytes IP header, 32 Bytes TCP header and 14 Bytes Ethernet header. Once concrete values r and β are known, the Payload Extractor transforms S^{obs} into S^{PL}.

Algorithm 1 is used to infer S^{PL} from S^{obs}. Let $Pr^{obs}(x)$ be the probability (according to S^{obs}) that the size of a flow is x and $Pr^{PL}(x)$ the probability (according to S^{PL}) that the size of a payload flow is x. $ACK(x) = 66 \cdot \left\lceil \frac{x}{MSS \cdot r} \right\rceil$ is the size of an ACK flow acknowledging the receipt of x payload bytes. MSS is the maximum segment size which in our setup was 1448. To convert S^{obs} into S^{PL}, the algorithm iterates over all flow sizes in descending order and removes the corresponding ACK flow from S^{PL}. This works because it always holds that the ACK-flow size is smaller than or equal to the corresponding payload-flow size.

4.3. Traffic Matrix Generator

From the outputs of the Deconvolver (B^{PL}_{intra} and B^{PL}_{inter}) together with the observed distribution of intra-rack communication partners per node for the intra-rack case (N^{obs}_{intra}) and the observed distribution of inter-rack communication partners per node for the inter-rack case (N^{obs}_{inter}), the *Traffic Matrix Generator* creates a $TM^{(PL)}$. This $TM^{(PL)}$ specifies payloads such that, when exchanged using TCP, this results in a $TM^{(gen)}$ having the same statistical properties as $TM^{(obs)}$. Matrix generation is explained in detail in Section 6. After the $TM^{(PL)}$ has been calculated, the payloads exchanged between any pair of hosts are divided into single payload flows.

4.4. Flowset Creator

Flows are generated by the *Flowset Creator*. The Flowset Creator gets S^{PL} from the Payload Extractor, IAT^{obs}, and a target traffic volume (which is the sum over all entries of the $TM^{(PL)}$ generated in the previous step). It outputs a set of flows whose flow sizes sum up to the target traffic volume.

To this end, the Flowset Creator creates payload flows with sizes distributed according to S^{PL}. When the payload flows are transferred over the network, the generated Layer 2 flows have to comply with IAT^{obs}. For this task, we need to infer IAT^{PL} from IAT^{obs} such that IAT^{PL} and IAT^{ACK} result in IAT^{obs}. However, the resolution of the data provided by [1] (or any other sources we are aware of) for IAT^{obs} is so low that we could not draw any conclusions on IAT^{PL}. This is why we use IAT^{obs} as an approximation for IAT^{PL}; this is a rough approximation, however, we do not have any better data at hand.

The generated flows only add up to the target traffic volume if IAT^{PL} and S^{PL} are chosen such that the sum of all generated flow sizes matches the traffic volume of the generated $TM^{(PL)}$. If this is not the case, we scale the inter-arrival times by a linear factor to generate more or less flows depending on the situation.

4.5. Mapper

In the last step of our traffic generator the flows generated by the Flowset Creator are mapped to the source-destination pairs specified by the $TM^{(PL)}$ as computed by the Traffic Matrix Creator. This mapping is done by the *Mapper* which uses a newly developed assignment strategy. The Mapper and our mapping strategy are explained in detail in Section 7.

5. Deconvolving traffic matrix entries

5.1. Problem description

The outcome of the traffic generation process is a schedule of payload transmissions specifying when which amount of payload is sent from one machine to another. In TCP, whenever a certain amount of payload is transferred over the network, this payload flow causes a second flow called ACK flow. The ACK flow acknowledges the correct reception of the payload flow but does not transmit any payload itself.[2] However, it adds traffic to the network. The traffic seen on Layer 2 is the sum of the payload flows and the ACK flows. We only have information from the observed $TM^{(obs)}$ but we want to build the inferred $TM^{(PL)}$. To this end, we need to compute B^{PL}_{intra} from B^{obs}_{intra} and B^{PL}_{inter} from B^{obs}_{inter}. Or, put in other words, we need to infer the Layer 4 distributions of non-zero TM entry sizes from the corresponding Layer 2 distributions. This section shows how to do that.

The individual non-zero traffic matrix entry sizes of a $TM^{(obs)}$ can be expressed as random variables $Z = X + Y$ where X and Y specify the amount of outgoing payload Bytes (X) and the amount of outgoing ACK Bytes (Y). The distribution of Z is given as B^{obs}_{inter} resp. B^{obs}_{intra} and it is the *linear convolution* of the distributions of X and Y, which we neither know.

When assuming that the ratio between payload packets and ACK packets is a constant r and by ignoring the facts that (a) the TCP protocol adds overhead to each single packet and (b) TCP uses additional messages for establishing and terminating sessions (TCP handshake), we can write Z as

$$Z = X + \beta Y$$

where $\beta = \frac{|ACK|}{|PAY|} \cdot \frac{1}{r}$ as before. We treat X and Y as independent and identically distributed (iid) random variables although X and Y might

[2] Data exchange between two hosts over different TCP connections is of course supported by DCT²Gen.

be correlated. However, we assume that this correlation is very low for the kind of software that runs in a data center.

RFC 1122 [14] states that for TCP it is not required to acknowledge the correct receipt of every single payload packet. Instead, one ACK can acknowledge multiple payload packets at once. This is called *delayed ACK*. However, the acknowledgement of payload must not be arbitrarily delayed. According to RFC 1122, the delay must be less than 0.5 seconds and there has to be at least one ACK packet for every second payload packet. Unfortunately, TCP implementations in modern Operating Systems do not follow this specification strictly. In experiments with Linux Kernel 3.11, e.g., the number of outstanding unacknowledged payload packets ranged up to 16 for a backlogged 1 GByte flow over a 1 Gbit/s link.

In the following, we show that for the TCP connections transferring most Bytes in a data center, the ratio between payload packets and ACK packets is on average $r \approx 2.5$.

5.2. Estimating payload to ACK ratio

We now show that r is nearly constant in our data-center scenario. Clearly, the value of r depends on (a) the available link speeds, (b) the TCP implementation, and (c) the distribution of flow sizes. We want to calculate r for payload-flow sizes distributed according to S^{PL}. To determine r, we have to compute a $TM^{(PL)}$ and divide its non-zero entries into payload flows. Then, this traffic can be emulated using a network emulator and the resulting r value can be observed. However, we know neither $TM^{(PL)}$ nor S^{PL}. To compute both we need to know r first, which means we are stuck in a vicious circle.

A pragmatic way of breaking the circle is to use $TM^{(obs)}$ as an approximate for $TM^{(PL)}$, divide the non-zero entries into payload flows distributed according to S^{obs} and emulate this traffic to determine r. This of course has a negative influence on the accuracy of the estimated r value. However, since (a) there is no reason why the traffic matrix should have a large effect on r (as long as the payload sizes stay the same), and (b) S^{obs} and S^{PL} are not too way off, the introduced error will be acceptable.

To estimate r we calculate a $TM^{(obs)}$, generate TCP traffic with payload sizes distributed according to S^{obs}, and emulate this traffic using a network emulator. In a subsequent step we analyze the generated ACK packets to approximate r.

We generated 60 s of TCP traffic for a data center consisting of 1440 servers organized in 72 racks of 20 servers each, interconnected in a Clos-like topology (for details, see Section 8). We emulate this data center with MaxiNet [15] which is a distributed Mininet version. We use a time dilation factor of 300 on a cluster of 12 servers equipped with Intel Xeon E5506 CPUs running at 2.16 Ghz.

On both emulated core switches we used `tcpdump` to write a trace of all packets passing the first interface.[3] In a subsequent step we analyzed all ACK flows in the trace to determine the ratio between the transferred payload and the number of ACK packets. Fig. 4 plots the number of payload Bytes (on TCP level) acknowledged by each ACK packet against the size of the 429,491 identified payload flows. The two horizontal lines mark 2896 Bytes and 4344 Bytes, which is one ACK packet for every second resp. every third payload packet (we used an MTU of 1500 which means the MSS was 1448). It can be seen that for each flow larger than $2^{16} \approx 65$ KB the ratio between payload packets and ACKs is between 2 and 3. For larger flows the ratio stabilizes at 2.5.

Note that the observable TCP dynamics depend on various environment characteristics. TCP always adapts to the current network situation by delaying ACKs and by enlarging or decreasing the TCP window size resulting in different data rates for the single flows.

[3] For performance reasons, it was not possible to create traces at all switches or interfaces. So we decided to use the core switches to be able to see traffic from all parts of the network.

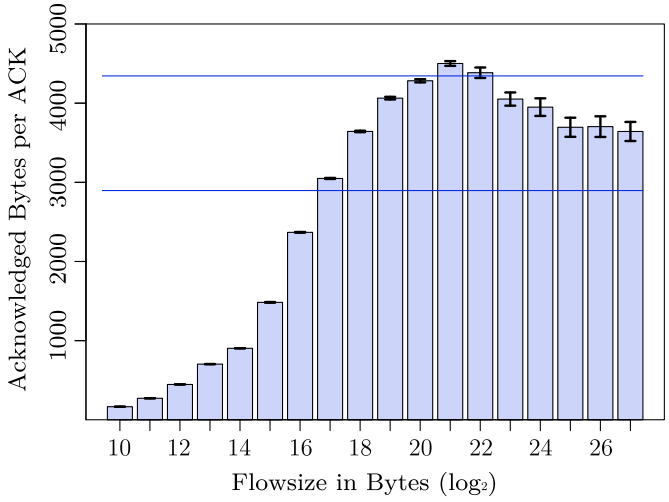

Fig. 4. Number of acknowledged payload Bytes per ACK packet plotted over different flow sizes. Error bars show confidence intervals of level 95%.

Thus, the ratio between payload packets and ACKs we found for our scenario can differ from the ratio in other network setups. In that case a different r value has to be given to DCT^{2}Gen.

5.3. Deconvolving TCP traffic

Once we know r, which tells how much overhead is created by the ACK packets, we can easily calculate β as it only depends on the MTU. Using the results of [16], we are now going to show how to extract the distribution of X from

$$Z = X + \beta Y$$

when β and the distribution of Z are known and X and Y are independent and identically distributed (iid). This result can then be used to to infer both B^{PL}_{intra} from B^{obs}_{intra} and B^{PL}_{inter} from B^{obs}_{inter}.

Let $f(t)$ denote the characteristic function of X which we want to calculate. Since the characteristic function of the sum of two independent random variables is the product of both their characteristic functions, we can write the characteristic function $g(t)$ of Z/β as

$$g(t) = f(t)\, f(\gamma t)$$

where $\beta = \frac{|ACK|}{|PAY|} \cdot \frac{1}{r}$ and $0 < \gamma = \frac{1}{\beta} < 1$. Due to [16], we can write

$$f(t) = \prod_{k=0}^{\infty} \frac{g(\gamma^{2k} t)}{g(\gamma^{2k+1} t)} \,.$$

Evaluating $f(t)$ on each point from the range of $g(\cdot)$ yields an approximation of the characteristic function of X. From the characteristic function, the density can be calculated by an inverse Fourier transformation.

5.4. Results

To ascertain that the deconvolution yields reasonable results, we now show the results of the deconvolution of B^{obs}_{inter} (as reported in [1]). We set r to 2.5, thus $\beta = \frac{66}{MSS} \cdot \frac{1}{2.5}$ and compute the deconvolution to retrieve B^{PL}_{inter}.

Then, we compute the implied B^{ACK}_{inter} based on B^{PL}_{inter} (Fig. 2). The function $ACK(p)$ is used to compute the size (in bytes) of an ACK flow corresponding to a payload flow of size p:

$$ACK(p) = 66 \cdot \left\lceil \frac{p}{MSS \cdot r} \right\rceil$$

where in our setup MSS was set to 1448. 66 is multiplied because in TCP an ACK packet has a size of 66 bytes. We calculate the resulting

Please cite this article as: P. Wette, H. Karl, DCT^{2}Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

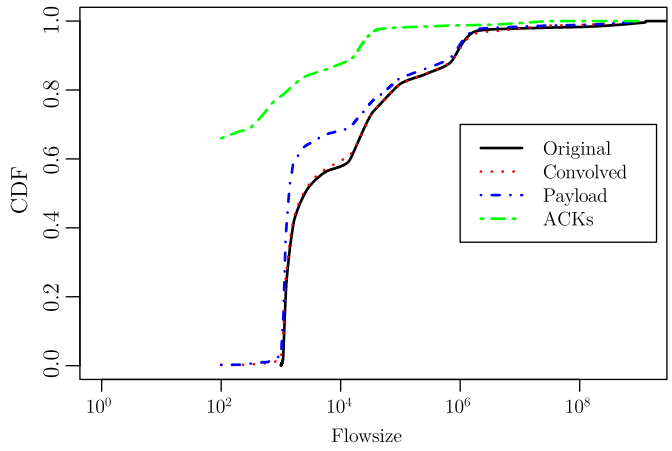

Fig. 5. Result of the deconvolution operation. The solid line shows B^{obs}_{inter} which is decomposed into B^{PL}_{inter} and the B^{ACK}_{inter}. The dotted line depicts the convolution of B^{PL}_{inter} and B^{ACK}_{inter}.

B^{gen}_{inter} as the convolution of B^{PL}_{inter} and B^{ACK}_{inter} and compare it to B^{obs}_{inter} to ascertain that the deconvolution was successful.

Fig. 5 shows the result of the deconvolution operation. It depicts the following four CDFs: (a) B^{obs}_{inter} as the original CDF, (b) B^{PL}_{inter} as the inferred payload-size distribution, (c) B^{ACK}_{inter} as the implied ACK-size distribution, and (d) B^{gen}_{inter} as the convolution of both B^{PL}_{inter} and B^{ACK}_{inter}. One can see that the CDFs of the original and the derived Layer 2 distribution are almost identical which shows that the deconvolution was successful.

One should note that the deconvolution only yields an approximation of B^{PL}_{intra} and might be noisy depending on the resolution of the input data. In our case, we extracted B^{obs}_{inter} from a log-scaled figure published in [1] which has a very bad resolution. The resulting noise is even more amplified by the transformation from the characteristic function to a probability density function. We thus had to perform some manual filtering on the density function to retrieve the function depicted in Fig. 5. This filtering basically removed negative values, smoothed the function, and scaled it to sum up to 1. We suspect that with proper data for B^{obs}_{inter}, this manual filtering is not necessary.

6. Generating traffic matrices

In this section we present our approach to generate traffic matrices. The mix of applications run in a data center has a strong influence on the resulting communication structure (and thus on the TMs). If only one single application were running, its communication pattern would be reflected by the resulting traffic matrix. But with more and more applications running simultaneously, the impact of any single application on the overall traffic will reduce and the aggregate behavior will appear more and more random. Moreover, as we stated in the introduction, we do not necessarily have the knowledge which applications are in fact running from the observed and available data. Therefore, we aim at generating TMs that express such (perhaps only seemingly) random traffic patterns. We do highlight that with the techniques developed in [13], DCT^{2}Gen can be extended to include application-specific communication patterns into the TM generation process. [13] analyzes packet traces to construct so-called *traffic dispersion graphs* modeling the communication structures of different applications between a set of end hosts. However, this requires access to packet traces from the network in question which are not available to us.

We generate a $TM^{(PL)}$ that specifies the amount of payload exchanged between server pairs within a fixed period. When this payload is transported using TCP, this generates a traffic matrix $TM^{(gen)}$ on Layer 2. For this TM it holds that:

- B^{gen}_{intra} equals B^{obs}_{intra}
- B^{gen}_{inter} equals B^{obs}_{inter}
- N^{gen}_{intra} equals N^{obs}_{intra}
- N^{gen}_{inter} equals N^{obs}_{inter}

To create a $TM^{(PL)}$, we first determine each node's number of inter- and intra-rack communication partners by computing a random variable from the corresponding distributions. Then, we use the numbers as node degrees and look for such a undirected simple graph[4] G. Finding a graph with a given inter- and intra-rack node degree is the *k-Partite Degree Sequence Problem* which is a variant of the intensively studied *Degree Sequence Problem*. We give an Integer Linear Program (ILP) to solve the k-Partite Degree Sequence Problem and study its run-time behavior.

In a subsequent step we transform the adjacency matrix of G into a traffic matrix by computing a random variable for the traffic volume for each edge using B^{PL}_{inter} resp. B^{PL}_{intra}.

6.1. Problem formalization

The problem of creating traffic matrices for n nodes with given intra- and inter-rack node degrees can be formalized as follows: The inter-rack node degree of a node is defined as the number of edges to nodes in different racks whereas the intra-rack node degree of a node is defined as the number of edges to nodes in the same rack. Let $V = \{v_1, v_2, \ldots, v_n\}$ be a set of vertices organized in racks of size m where $v_{i \cdot m}, v_{i \cdot m+1}, \ldots, v_{(i+1) \cdot m-1} \forall\, 0 \leq i < \lceil n/m \rceil$ are located in the same rack i. Let $D^{int} = (d^{int}_1, d^{int}_2, \ldots, d^{int}_n)$ be the desired intra-rack node degrees and $D^{ext} = (d^{ext}_1, d^{ext}_2, \ldots, d^{ext}_n)$ the desired inter-rack node degrees for all n nodes. We are looking for an undirected simple graph $G = (V, E)$ where the node degrees follow the intra- and inter-rack degrees given by D^{int} and D^{ext}.

6.2. The degree sequence problem

Let $G = (V, E)$ be a simple graph on n vertices. We call the decreasing order of the node degrees of V the *degree sequence* of G.

Problem 1 (The Degree Sequence Problem)**.** Let $V = (v_1, v_2, \ldots, v_n)$ be a set of nodes and $D = (d_1, d_2, \ldots, d_n)$, $d_i \geq d_{i+1}\ \forall\, 0 < i < n$, the desired node degrees. Find a simple graph $G = (V, E)$, $E \subseteq V \times V$, where the degree sequence of G is equal to D. If such a graph exists, D is called *realizable*.

Problem 1 is extensively studied [17,18]. The main results on the Degree Sequence Problem for simple graphs are Theorem 1 and Theorem 2.

Theorem 1 (Erdös–Gallai)**.** *D is a realizable degree sequence for a simple graph on n nodes if and only if*

1. *The sum of all desired node degrees is even*
2. $\sum_{i=1}^{k} d_i \leq k(k-1) + \sum_{i=k+1}^{n} \min(d_i, k)\ \forall\, 1 \leq k \leq n$.

Theorem 2. *$D = (d_1, d_2, \ldots, d_n)$ is realizable as a simple graph if and only if $D' = (d_2 - 1, d_3 - 1, \ldots, d_{d_1+1} - 1, d_{d_1+2}, d_{d_1+3}, \ldots, d_n)$ is realizable as a simple graph.*

From Theorem 2 the iterative algorithm stated in Algorithm 2 can be deduced to create a simple graph with a given node degree. The algorithm creates a graph for which it holds that $\deg(v_i) = d_i$ (if such a graph exists).

[4] A simple graph is an undirected graph where no node has an edge to itself and no more than one edge between the same pair of nodes exists.

Algorithm 2 ConstructGraph ($D = (d_1, \ldots, d_n)$).

1: $G = (V, E), V = \{1, 2, \ldots, n\}, E = \{\}$
2: Let the initial residual node degree of node v_i be d_i.
3: Let $U = (u_1, u_2, \ldots, u_n)$ be the list of vertices decreasing in the order of their residual node degree.
4: Create edges between u_1 and the next d_1 nodes in U.
5: **if** no d_1 nodes exists with residual node degree > 0 **then**
6: **return** Error
7: **end if**
8: Update D and corresponding U as stated by Theorem 2.
9: If U is not empty, goto 4.
10: **return** $\Pr'(\cdot)$

6.3. The k-Partite Degree Sequence Problem

Creating inter-rack edges is different from Problem 1 because here there exist sets of nodes between which no edges are permitted. These sets are the sets of nodes located in the same rack. This leads us to Problem 2, called the *Degree Sequence Problem on k-Partite Graphs*.

Problem 2. (Degree Sequence Problem on k-Partite Graphs) Given k degree sequences $D_1, D_2, \ldots, D_k$, find an undirected k-Partite Graph G where each partition i consists of $|D_i|$ nodes and for each node v in partition i it holds that $\deg(v) = D_{i_v}$. We call $D_1, D_2, \ldots, D_k$ *realizable* if such a graph exists.

Problem 2 is a special case of the *Restricted Degree Sequence Problem* [19] in which arbitrary edges are forbidden to use. The best known algorithm to solve this problem requires to find a perfect matching on a simple graph of $\Omega(n^2)$ nodes. This makes this approach inapplicable to our problem: It already took more than 36 minutes on an Intel i7 2.2 Ghz processor to calculate a graph for seven racks (each consisting of 20 servers) using the Boost graph library.

Reference [20] presents the *Degree Sequence Problem with Associated Costs* where the goal is to find a minimum cost realization of a given degree sequence. It is possible to model Problem 2 when setting all costs for intra-rack edges to infinity. However, the running time to solve the problem is also dominated by finding a perfect matching on a graph with $\Omega(n^2)$ nodes. We thus model our problem as an ILP with n constraints, which is faster to solve for the problem instance sizes in this context.

ILP 1 models Problem 2 where $D^{\text{ext}} = D_1 \cup D_2 \cup \ldots \cup D_k$. In case that D^{ext} is realizable, ILP 1 computes a graph G^{ext} with degree sequence D^{ext}. If D^{ext} is not valid it will compute the Graph G^{ext} which has the highest possible edge count under the condition that no node has a higher degree than specified by D^{ext}.

ILP 1. (Constructing an Inter-Rack Graph)

$$\text{maximize} \quad \sum_{0<i<n} \sum_{0<j<i} b_{i,j} \qquad b_{i,j} \in \{0, 1\}$$

w.r.t.

$$\sum_{j \in \text{inter}(i)} b_{\max(i,j),\min(i,j)} \leq d_i^{\text{ext}} \qquad \forall 1 \leq i \leq n$$

In ILP 1, $b_{i,j}$ equals 1 if an undirected edge exists between nodes i and j. Note that ILP 1 only models the lower triangular matrix of the adjacency matrix of G^{ext} because G^{ext} is undirectional. inter(i) describes the set of nodes that are not in the same rack as node i and is defined as

$$\text{inter}(i) = \left\{ j \mid \left\lfloor \frac{j}{m} \right\rfloor \neq \left\lfloor \frac{i}{m} \right\rfloor \forall\, 0 < j \leq n \right\}.$$

We use ILP 1 to compute G^{ext}. To show that the running time is acceptable for practical instances we are conducting the following experiment on an Intel i7 960 CPU running at 3.2 GHz with 24 GB of DDR3 memory. The ILP 1 is solved using Gurobi 5.6. We generated problem sequences D^{ext} for different number of nodes n and a fixed rack size $m = 20$. The single d_i^{ext}'s are drawn uniformly at random from the set $\{0, 1, \ldots, \lfloor n \cdot p \rfloor\}$ for values of $p \in \{0.05, 0.1, 0.2, 0.3\}$. For each parameter pair (n, p) we solved 40 problem instances and measured the running times. Fig. 6 plots the required running time against the size of the problem instance n and the different values for p. One can see that the running time is exponential in n and that the number of edges (controlled by parameter p) has no significant influence on the running time.

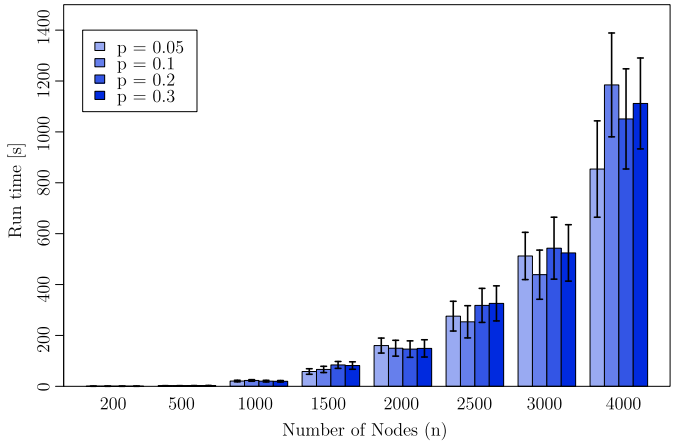

Fig. 6. Run time of ILP 1 for different problem sizes. Errorbars indicate the confidence intervals for a confidence level of 95%.

6.4. Generating a traffic matrix

The process of creating a traffic matrix can be divided into the two steps (a) finding the positions of non-zero traffic matrix entries, and (b) assigning traffic volumes to non-zero traffic matrix entries.

We are finding the positions of non-zero traffic matrix entries by creating a graph G with given intra- and inter-rack node degrees. To this end, two graphs G^{int} and G^{ext} are constructed. $G^{int} = (V, E^{int})$ only contains intra-rack edges with degree sequence D^{int} and $G^{ext} = (V, E^{ext})$ only contains inter-rack edges with degree sequence D^{ext}. We construct G by setting $G = \left(V, E^{int} \cup E^{ext}\right)$. Whenever there is an edge between a pair of nodes i and j we make (i, j) a random variable in the traffic matrix which is distributed according to $\text{B}^{\text{PL}}_{\text{inter}}$ resp. $\text{B}^{\text{PL}}_{\text{intra}}$.

To create G^{int}, each rack can be examined separately using Algorithm 2. This leads to k unconnected subgraphs which model the communication between servers in the same racks. However, this only works if the degree sequences are realizable. But, as we draw the degree sequences randomly from the given distribution and the rack sizes are relatively small, in most cases the demanded degree sequences are not realizable. This is problematic as we are not allowed to redraw the degree sequences in that case because this would lead to a wrong distribution of intra-rack node degrees. Note that this problem is specific to the intra-rack case where the number of nodes is very small and the demanded node degrees are very high. For the inter-rack case, the probability of sampling a non-realizable degree sequence is much lower because of the large number of nodes and the relatively small demanded node degrees.

To compute intra-rack edges with degrees following $\text{N}^{\text{obs}}_{\text{intra}}$, we developed ILP 2. ILP 2 assigns penalties to each node in case it does not meet its demanded node degree. The penalty of a node is defined as the absolute difference between the demanded node degree (given by D^{int}) and the node degree in the solution calculated by the ILP

Please cite this article as: P. Wette, H. Karl, DCT²Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

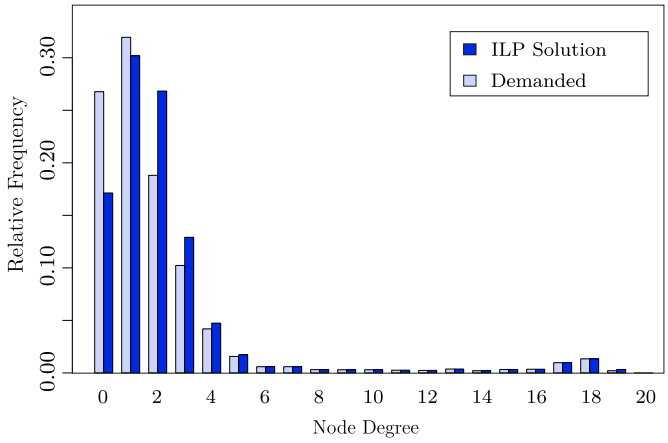

Fig. 7. Comparison of (a) the distribution of node degrees in the demanded degree sequence and (b) the distribution of node degrees of the solution computed by ILP 2.

itself. ILP 2 minimizes the sum over the penalties of all nodes i divided by the probability of degree d_i (according to N^{obs}_{intra}). This way, the sum of the *relative distances* between the degree distribution computed by the ILP and N^{obs}_{intra} is minimized.

ILP 2. (Constructing an Intra-Rack Graph)

$$\begin{aligned} & \text{minimize} && \sum_{0<i<n} \frac{p_i}{Pr(d_i)} && \\ \text{w.r.t.} & && && \\ & p_i && \geq && 0 && \forall 1 \leq i \leq n \\ & p_i && \geq && \sum_{j \in \text{intra}(i)} b_{i,j} - d_i && \forall 1 \leq i \leq n \\ & p_i && \geq && d_i - \sum_{j \in \text{intra}(i)} b_{i,j} && \forall 1 \leq i \leq n \end{aligned}$$

In ILP 2, p_i is the penalty assigned to node i. The demanded node degree of i is denoted d_i and $b_{i,j}$ is 1 if there is an edge between nodes i and j in the calculated graph. For practical instances, the run time of the ILP 2 is not critical as each rack can be examined separately and racks typically consist of up to 40 servers only.

Fig. 7 shows the solution quality of ILP 2 in an experiment with 10,000 servers organized in racks of 20 servers each. The 10,000 demanded node degrees are distributed according to N^{obs}_{intra}. One can see that the distribution of node degrees computed by ILP 2 is comparable to N^{obs}_{intra}. The distributions match very well for node degrees with small densities. For larger densities, the gap between the distribution of demanded degrees and the distribution of node degrees computed by ILP 2 is larger. However, the solution of ILP 2 is of sufficient quality for our purpose as can clearly be seen in our evaluation (Figs. 16 and 17).

7. Generating TCP flows

7.1. Overview

A traffic matrix computed by the Traffic Matrix Creator states the amount of bytes exchanged by node pairs in a fixed time. Data-center traffic consists mostly of short-lived flows [1,2]. Thus, for each communicating node pair (non-zero TM entry), the bytes have to be separated into different flows. We describe a Layer 4 flow as the 4-tuple *(start time, source, destination, size)*. This section only deals with payload flows.

Given a $TM^{(PL)}$ determining how many bytes to transfer between every pair of nodes, the question to answer is: How to separate the non-zero entries of a $TM^{(PL)}$ into flows such that flow sizes are following S^{PL} and the flow inter-arrival times are distributed according to IAT^{PL}?

Our strategy is to first generate a set of flows complying with S^{PL} and IAT^{PL} and afterwards map these flows to the non-zero entries of the $TM^{(PL)}$.

7.2. Generating flows

Generating flows complying with S^{PL} and IAT^{PL} 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 generate flows for them according to S^{PL} and IAT^{PL}. But this approach raises some questions, for example:

When to stop generating new flows for (u, v)***?***
We could stop assigning flows to (u, v) when the sum of flow sizes for (u, v) is larger than specified by the TM. But then, more traffic would be generated than is specified by the TM. Another way would be to stop generating flows for (u, v) when the next flow that is to be generated would exceed the amount stated by the TM. This way, the traffic generated by the flows would be less than specified by the TM. Hence, no matter how we decide, the resulting $TM^{(gen)}$would not follow B^{PL}_{inter} and B^{PL}_{intra}.

What if for a small TM entry a huge flow size is generated?
Generating a new flow size in this situation distorts the resulting flow-size distribution. And by assigning the too large flow, the resulting $TM^{(gen)}$would not comply with B^{PL}_{inter} and B^{PL}_{intra}.

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 S^{PL} and IAT^{PL} (where "unmapped" means the flow is not yet assigned to a source-destination pair, *s-d pair*). Afterwards, flows get mapped to 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 (and actually unlikely) that an exact mapping exists. Nevertheless, a good mapping strategy assigns flows such that the sum of flow sizes between nodes i and j is as close as possible to TM entry (i, j).

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 such that flow sizes sum up to s_M. We denote the sum of all generated flow sizes as s_F. As s_M is a random variable it will hold that $s_M = s_F \cdot \varepsilon$, $\varepsilon \in \mathcal{R}^+_0$, where ε is the imbalance factor between the size of the flows and the TM. Of course, ε should be very close to 1 (meaning there is no imbalance at all), which is why we start over to generate the *whole set* of unmapped flows with adjusted flow inter-arrival times as long as $|\varepsilon - 1| > 0.01$. This means that the sum of all generated flow sizes deviates at most 1% from the traffic specified by the TM. We assume this to be a reasonably small error.

We will now present two different strategies to map the unmapped flows to node pairs. The first one is a purely random process and the second one uses a variation of the queuing strategy *deficit round robin* (DRR) [21]. Afterwards, we study the quality of both strategies.

The randomized assignment uses the TM as a probability distribution and, for each generated flow, draws a node pair from this distribution. In this process, we define the initial probability to assign a flow to node pair (i, j) as the TM entry (i, j) divided by s_M. After a flow has been assigned, the probability distribution at the point of the node pair is lowered proportionally to the size of the flow.

Please cite this article as: P. Wette, H. Karl, DCT²Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

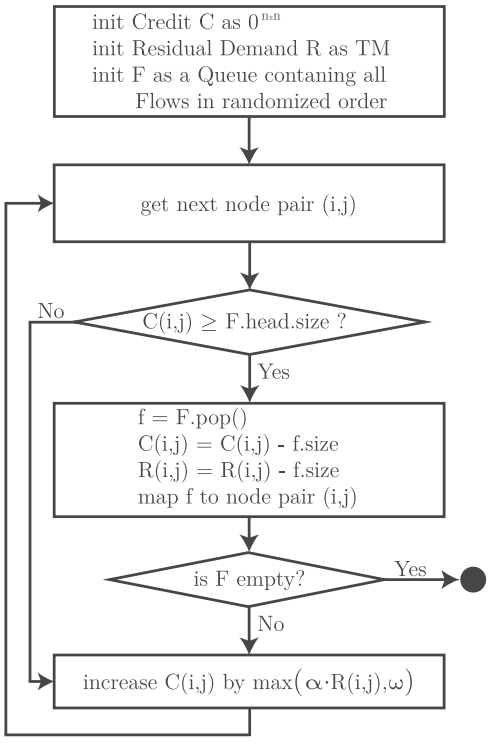

Fig. 8. Deficit Round Robin inspired algorithm for selecting s-d pairs for flows.

The second strategy is inspired by DRR. DRR schedules jobs of different sizes and classes onto a shared processor. The goal of DRR is to share the processor among all classes according to the ratio of their priorities. To this end, each class is assigned a *priority* and a *credit*. DRR loops Round Robin through all classes. In each iteration of the loop, the credit of each class is raised by some constant (called quantum) weighted by the priority of the class. If for a class there exists a job with a size smaller than the current credit of the class, this job is scheduled to the processor and the credit of the class is lowered by the size of the job.

We use a DRR variant to map flows to node pairs. In this variant, node pairs correspond to classes and flows correspond to jobs. The only difference in our variant is that we do not schedule flows onto a shared processor; we schedule flows on node pairs. The priority of a node pair is proportional to the size of its residual traffic matrix entry. We loop Round Robin over all node pairs and raise their credit proportional to their residual TM entry. Whenever the unmapped flow under consideration is smaller then or equal to the credit of the node pair, this flow is mapped to the node pair and the credit is lowered accordingly.

Our adapted version of the DRR strategy can be seen in Fig. 8. In this algorithm, i always corresponds to a source, j to a destination and R is the *residual* traffic between i and j as specified by the TM: whenever a flow is assigned to (i, j), $R_{i,j}$ is decreased by the size of the flow. F is a queue that initially contains all flows in a randomized order. $C_{i,j}$ is the credit (akin to DRR) of the node pair (i, j). $C_{i,j}$ is decreased whenever a flow is assigned to (i, j) 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 (i, j) as no valid candidate has been found. (i, j) is a valid candidate for flow f if $C_{i,j}$ is larger than or equal to the size of f. After a pair (i, j) has been inspected its deficit counter is increased by $\max(\alpha \cdot R_{i,j}, \omega)$; α and ω 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 $\omega = 100$ cause no significant improvement of 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.

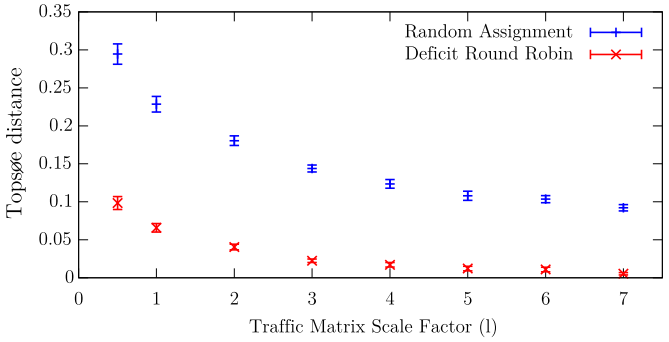

Fig. 9. Topsøe distance of flow assignment methods over different traffic volumes. Error bars show confidence intervals for a confidence level of 95%.

7.3. 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. In reality, we will produce a traffic matrix with slight derivations. To express the difference between the given 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 *Kullback–Leibler divergence* (KL), but KL requires 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* (Eq. 7.1) [22] does not require this implication and can be used instead to compute the distance between two probability distributions.

$$\text{Topsøe}(M, M') = \sum_{(i,j)} \left(M_{i,j} \ln \frac{2M_{i,j}}{M_{i,j} + M'_{i,j}} + M'_{i,j} \ln \frac{2M'_{i,j}}{M_{i,j} + M'_{i,j}} \right) \tag{7.1}$$

We look at the Topsøe distance for *different* load levels of a network because 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. A load level is created by multiplying the TMs with a factor l; we denote the corresponding TM by lM. We then assign flows for lM to s-d pairs and calculate the TM $(lM)'$ based on that flow assignment.

We use lM as the ground truth and express the difference between lM and $(lM)'$ as the relative entropy of both matrices. Fig. 9 shows the relative entropy obtained via either the random strategy or the Deficit Round Robin strategy calculated as averaged over the *Topsøe distance* of 40 matrices of 10 s generated traffic each $\left(\frac{1}{40}\sum_{i=1}^{40} \text{Topsøe}\,.(lM, (lM)')\right)$. The data center for which we generate traffic consists of 75 racks with 20 servers each. It is the same size that was used in the study [1]. 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. We will only consider the DRR-based scheme henceforth.

8. Empirical evaluation

8.1. Approach

DCT^{2}Gen works properly if it is able to compute a schedule of TCP payload transmissions where (when transferred over a network)

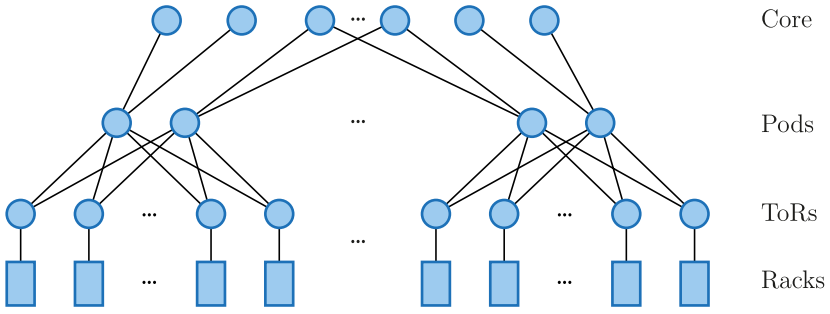

Fig. 10. Sketch of the Clos-like topology that was used in our experiments.

(a) the generated $TM^{(gen)}$ has the same properties as $TM^{(obs)}$ and (b) the generated flows have the same properties as the observed flows.

We use a stochastic analysis of the generated TCP schedule to confirm that $TM^{(gen)}$ follows the same probability distributions as $TM^{(obs)}$. To this end, we compute N^{gen}_{intra}, N^{gen}_{inter}, B^{gen}_{intra}, B^{gen}_{inter}, and IAT^{gen} based on the Layer 4 schedule and compare them to N^{obs}_{intra}, N^{obs}_{inter}, B^{obs}_{intra}, B^{obs}_{inter}, and IAT^{obs}. A network emulation through MaxiNet is used to capture the effects of TCP when the generated traffic is replayed on a data-center topology. From the results of the emulation, we compute S^{gen} and compare them to S^{obs}.

8.2. Traffic properties used in the evaluation

According to [1], N^{obs}_{intra} and N^{obs}_{inter} are heavy-tailed in typical data centers. It is reported 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 either exchange only a small amount or a large amount of data, whereas traffic across racks is either small or medium per server pair.

Kandula et al. [1] 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 [2] looked at traffic from 10 different data centers. They showed that across all 10 data centers S^{obs} is nearly the same. 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.

For evaluation, we used the observed distributions by [1,2] as an input to our traffic generator. Both studies reason about all the traffic in data centers. In addition to traffic transported with TCP, this includes ARP, DNS and many more protocols that do *not* use TCP for transport. This results in traffic characteristics that cannot be reproduced using TCP only. A flow resulting from an ARP request, for example, has a size of 60 Bytes which was also the smallest reported flow size. Due to the three-way handshake used to establish and tear down TCP sessions the smallest possible flow size (on Layer 2) TCP can produce is 272 Bytes. For evaluation we increased all flow sizes by 212 Bytes to remove this mismatch.

As a result of that increase, B^{obs}_{intra} and B^{obs}_{inter} no longer match the enlarged S^{obs}. This makes it impossible to have a good flow assignment because there are not enough small flows to be mapped to the small non-zero TM entries. To counteract this, we increased B^{obs}_{intra} and B^{obs}_{inter} by 1000 Bytes.

Note that the performed changes are only minor. The average flow size extracted from [2] is 142 KB. Thus, increasing the size of each flow by 219 Bytes is an increase of 0.15 % on average. The average non-zero intra-rack traffic matrix entry has a size of 12.6 MB, the average non-zero inter-rack traffic matrix entry 12.4 MB. Thus, an increase of 1000 Bytes per non-zero traffic matrix entry is negligible (about 0.1 %).

8.3. Topology and emulation environment

To include the effects of TCP into our evaluation, we choose to emulate a data center consisting of 72 racks employing a *Clos-like topology*. From the emulation, we are able to determine S^{gen} and IAT^{gen}. A sketch of the emulated topology can be seen in Fig. 10. Each rack consists of 20 servers and one ToR switch, which makes 1440 servers overall. Servers are connected by 1 Gbit/s links to ToR switches. Pods consist of *eight* ToR switches which are connected to *two* pod switches with 10 Gbit/s links. Pod switches are connected to two *core switches* with 10 Gbit/s links. The core layer in our topology consists of *two* switches. We assume a forwarding delay of 0.05 ms per switch. In each experiment, we emulated 60 s of traffic. This traffic was generated from the statistics reported in the previous section. We used a time dilation factor of 200, which means one experiment completed after 200 min.

For emulation, we used 12 physical worker nodes equipped with Intel Xeon E5506 CPUs running at 2.16 GHz, 12 Gbytes of RAM and 1 GBit/s network interfaces connected to a Cisco Catalyst 2960G-24TC-L Switch. Routing paths are computed using equal cost multipath (ECMP) implemented on the Beacon controller platform [23]. 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 whole 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 likely not keep up with the high flow arrival rates; for details see [15].

8.4. Results

To verify that DCT^{2}Gen produces a reasonable flow schedule, the traffic created by the schedule (box 5 in Fig. 1) must have the same properties as the observed traffic (box 1 in Fig. 1). As (a) we do not have access to the observed Layer 2 traces and (b) it is unclear how to directly compare two packet traces with each other, we compare the statistical properties of the two traces with each other (boxes 2 and 6 in Fig. 1). Comparison is done throughout the following sections where each statistical property is inspected individually. Due to the huge amount of samples (our collected packet traces contain 7,060,194 flows, 330,155 distinct intra-rack and 1,675,305 inter-rack TM entries; each of our 16 generated Layer 4 schedules contains around 6 million flows) it is not easily possible to use any goodness of

Please cite this article as: P. Wette, H. Karl, DCT^{2}Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

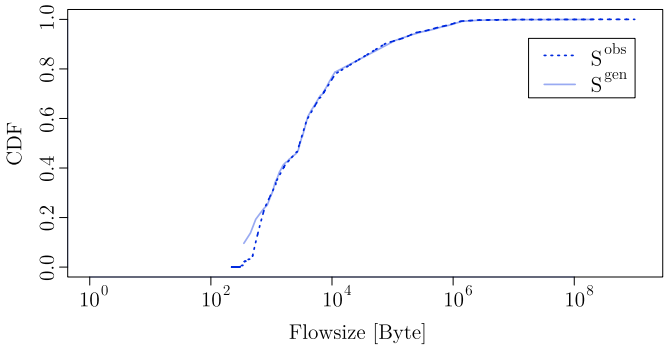

Fig. 11. Comparison between S^{gen} and S^{obs}.

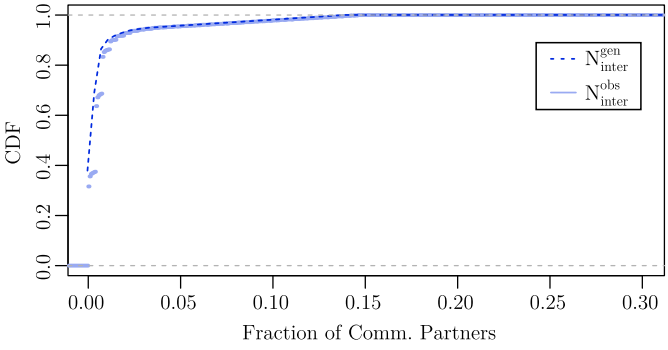

Fig. 12. Comparison between N^{gen}_{inter} and N^{obs}_{inter}.

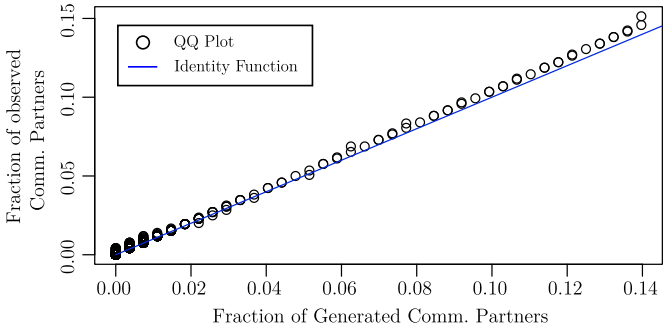

Fig. 13. QQ-plot of N^{gen}_{inter} and N^{obs}_{inter}.

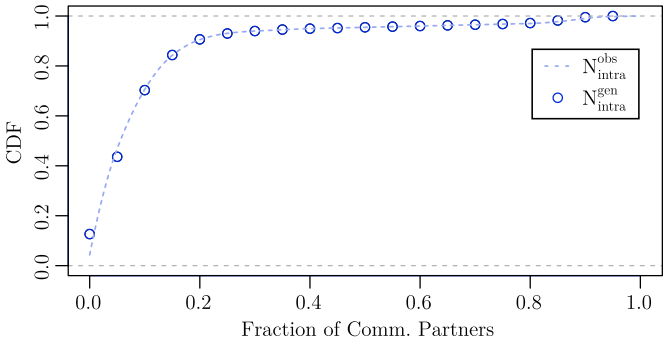

Fig. 14. Comparison between N^{obs}_{intra} and N^{gen}_{intra}.

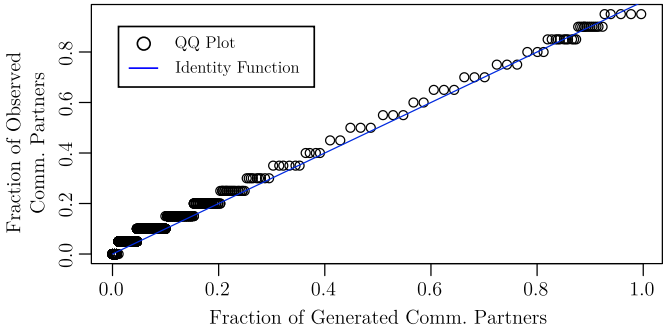

Fig. 15. QQ-plot of N^{obs}_{intra} and N^{gen}_{intra}.

fit test to judge whether the generated distributions match the corresponding observed distributions. This is because there exist small statistical differences between both distributions that together with the large set of samples are big enough for the goodness of fit tests to reject, but too small to be of practical importance for our purpose (these differences are statistically significant, but not relevant). We instead analyze the distributions by using Quantile-Quantile plots (QQ-plots)[5].

8.4.1. Generated flow-size distribution

To determine S^{gen} we emulated 60 s of data-center traffic consisting of 1440 hosts as described previously. A packet trace was captured on the first interface of each emulated core switch. We conducted 16 independent experiments (with 16 different Layer 4 schedules) and used the corresponding 32 traces to compute S^{gen}. The number of captured flows over all experiments is 7,060,194.

Fig. 11 plots S^{obs} and S^{gen}. It can be seen that the distributions clearly match for flow sizes larger 1000 bytes. The distributions of smaller flows, however, do not match well. We suspect this is partly due to the behavior of TCP and partly due to our assumptions on the size of ACK flows as most flows smaller than 1000 bytes are ACK flows (Fig. 5). As discussed in Section 5.2, smaller flows tend to have a lower ACK-to-payload ratio. The Flowset Creator, however, calculates the size of each induced ACK flow with a fixed ratio of r which results in the slightly wrong distribution of ACK-flow sizes. In the following subsections and figures, we use *comm.* as an abbreviation for *communication*.

8.4.2. Inter-rack comm. partners

To determine N^{gen}_{intra} and N^{gen}_{inter} we used the same 16 traffic schedules as before. Fig. 12 plots the cumulative density function of the number of communication partners per server for both N^{gen}_{inter} and N^{obs}_{inter}. This number is normalized to the total number of servers in the data center, i.e., a value of 1 means communication with all servers in the data center and a value of 0 means no communication at all. From the plot no difference between the two distributions is discernible. The corresponding QQ-plot (Fig. 13) also does not show any significant differences between N^{obs}_{inter} and N^{gen}_{inter}.

8.4.3. Intra-rack comm. partners

The comparison between N^{obs}_{intra} and N^{gen}_{intra} (Fig. 14) shows that our generated traffic contains a little too many intra-rack communication partners with a low degree. Despite that, both CDFs are nearly identical. This can also be confirmed by looking at the corresponding QQ-Plot (Fig. 15). The plot shows an almost straight line that lies a bit above the identity function. This result is in line with what is discussed in Section 6.4.

8.4.4. Intra-rack traffic

The $TM^{(obs)}$s used in this section are deduced from the same 16 traffic schedules we used in Section 8.4.3. To compute the single traffic matrix entries, we fixed the payload-to-ACK ratio r to 2.5 (see Section 5.2) and computed the size of the flows on Layer 2

[5] QQ-plots are plotting the quantiles of both distributions against each other. If the plot shows the identity function, this is an indicator that the distributions fit [24].

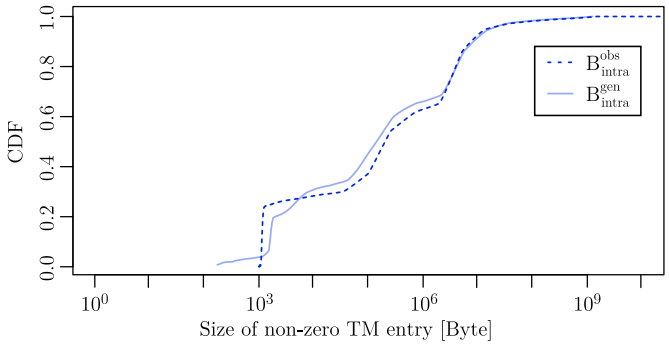

Fig. 16. Comparison between B^{obs}_{intra} and B^{gen}_{intra}.

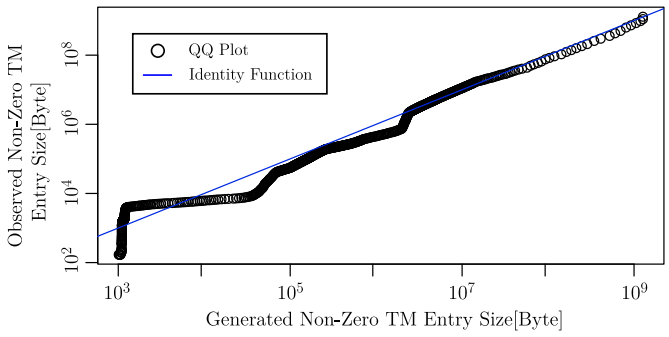

Fig. 17. QQ-plot of B^{obs}_{intra} and B^{gen}_{intra}.

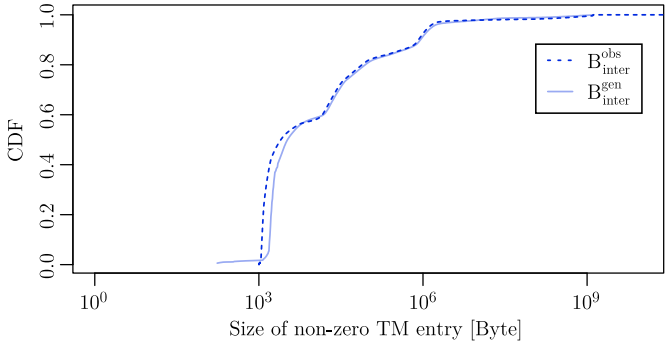

Fig. 18. Comparison between B^{obs}_{inter} and B^{gen}_{inter}.

between each pair of servers. From that, we calculated the respective 96 $TM^{(obs)}$s (each for a period of 10 s).

The corresponding B^{gen}_{intra} is compared to B^{obs}_{intra} in Fig. 16. Except for entries smaller than 10^4 Bytes, B^{gen}_{intra} is strictly following B^{obs}_{intra}. This can further be confirmed by the QQ-Plot (Fig. 17) which additionally only shows a small anomaly of the distribution for entries around 10^6 Bytes.

The difference between both distributions in the smaller entries is due to the process of mapping single flows to traffic matrix entries. The goal of the Mapper is to distribute flows to traffic matrix entries such that for each node pair the difference between their TM entry and the sum of flow sizes between that nodes is minimized per server pair. The smaller the TM entry, the fewer flows can be mapped onto the corresponding node pair which means it is harder to find a well fitting mapping.

8.4.5. Inter-rack traffic

B^{obs}_{inter} and B^{gen}_{inter} are plotted in Fig. 18; the corresponding QQ-plot can be seen in Fig. 19. From Fig. 18, we observe the same situation as in the *intra*-rack case. The QQ-plot additionally exposes differences for the distribution of large entries ($> 10^7$). This effect in the

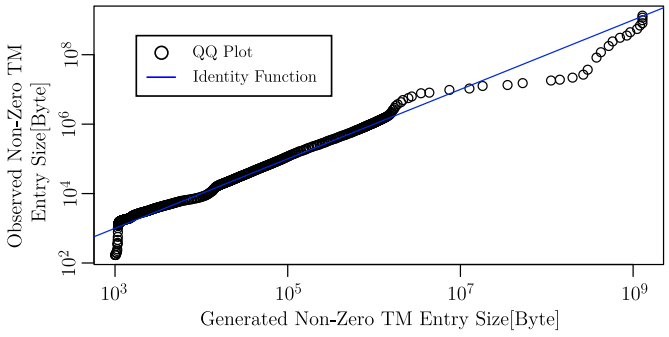

Fig. 19. QQ-plot of B^{obs}_{inter} and B^{gen}_{inter}.

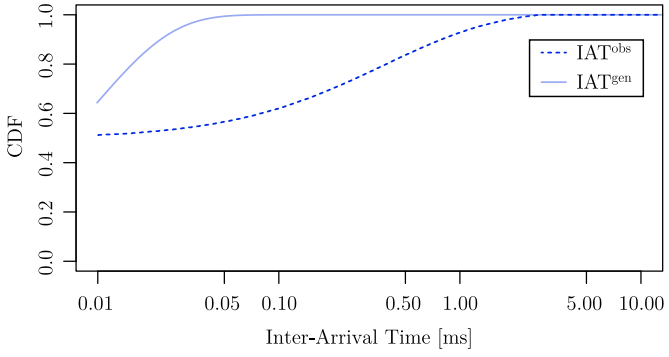

Fig. 20. Comparison between IAT^{obs} and IAT^{gen}.

QQ-plot is caused by only a slight difference between the tails of both distributions. As the tails of both B^{obs}_{inter} and B^{gen}_{inter} are very long, slight differences in the probabilities have a huge impact on the QQ-plot.

8.4.6. Flow inter-arrival time

To compute IAT^{gen}, we used the same 16 traffic schedules as before. In the Flowset Creator, IAT^{gen} is manipulated such that the bytes contained in all generated flows are matching the total traffic of the traffic matrix generated in the Traffic Matrix Generator. Both IAT^{gen} and IAT^{obs} can be seen in Fig. 20. Apparently, these distributions do not match. The reason for this mismatch is the manipulation done in the Flowset Creator. With IAT^{obs} extracted from [1] it was not possible to create enough flows to fill up the generated traffic matrices. This can have two causes: Either the IAT^{obs} reported in [1] does not match the used S^{obs} or the data provided in [1] has such a low resolution that we were not able to fully recover it. It would be interesting to repeat this work based on data with better quality.

9. Conclusion

The traffic generator DCT^{2}Gen presented in this work creates a Layer 4 traffic schedule for arbitrary sized data centers. When the scheduled payloads are transported using TCP, this produces Layer 2 traffic with properties that can be defined in advance using a set of probability distributions. Our evaluation showed that DCT^{2}Gen reproduces these properties with high accuracy. Solely the generated flow inter-arrival time distribution does not match our chosen target distribution. As DCT^{2}Gen manipulates the inter-arrival time distribution to adjust the amount of flows to the given traffic matrices, this is not surprising. We suspect that this difference will be significantly smaller when using input data of higher quality.

Given that DCT^{2}Gen generates a schedule of payload transmissions between all hosts in a data center it is suitable for simulations, network emulations, and testbed experiments. Using our generated traffic schedule combined with a large-scale network emulator such

Please cite this article as: P. Wette, H. Karl, DCT^{2}Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001

as MaxiNet, novel networking ideas can be evaluated under highly realistic conditions which brings new ideas a step closer to deployment in production environments.

Acknowledgment

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

References

[1] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, R. Chaiken, The nature of data center traffic: measurements & analysis, in: Proceedings of the 9th ACM SIGCOMM Conference on Internet Measurement Conference (IMC), 2009, pp. 202–208, doi:10.1145/1644893.1644918.
[2] T. Benson, A. Akella, D.A. Maltz, Network traffic characteristics of data centers in the wild, in: Proceedings of the 10th ACM SIGCOMM conference on Internet measurement, in: IMC '10, ACM, New York, NY, USA, 2010, pp. 267–280, doi:10.1145/1879141.1879175.
[3] Apache, Hadoop, (http://hadoop.apache.org) (accessed 16.12.15).
[4] S. Avallone, A. Pescape, G. Ventre, Analysis and experimentation of internet traffic generator, in: ACM Workshop on Models, Methods and Tools for Reproducible Network Research, 2004, pp. 70–75.
[5] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, G. Ventre, D-ITG distributed internet traffic generator, in: Proceedings of the First International Conference on the Quantitative Evaluation of Systems (QEST 2004), IEEE, 2004b, pp. 316–317.
[6] J. Laine, S. Saaristo, R. Prior, Real-time UDP Data Emitter, (http://rude.sourceforge.net) (accessed 16.12.15).
[7] K. V. Vishwanath, A. Vahdat, Swing: realistic and responsive network traffic generation, IEEE/ACM Trans. Netw. (2009) 712–725.
[8] C. Barakat, P. Thiran, G. Iannaccone, C. Diot, P. Owezarski, Modeling internet backbone traffic at the flow level, IEEE Trans. Signal Process. 51 (8) (2003) 2111–2124.
[9] J. Sommers, H. Kim, P. Barford, Harpoon: a flow-level traffic generator for router and network tests, in: Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems, 2004.
[10] S. Uhlig, B. Quoitin, J. Lepropre, S. Balon, Providing public intradomain traffic matrices to the research community, SIGCOMM Comput. Commun. Rev. 36 (1) (2006) 83–86, doi:10.1145/1111322.1111341.
[11] P.E. Heegaard, GenSyn - a java based generator of synthetic internet traffic linking user behaviour models to real network protocols, in: ITC Specialist Seminar on IP Traffic Measurement, Modeling and Management, 2000.
[12] M.C. Weigle, P. Adurthi, F. Hernández-Campos, K. Jeffay, F.D. Smith, Tmix: a tool for generating realistic tcp application workloads in ns-2, ACM SIGCOMM Comput. Commun. Rev. 36 (3) (2006) 65–76.
[13] P. Siska, M.P. Stoecklin, A. Kind, T. Braun, A flow trace generator using graph-based traffic classification techniques, in: Proceedings of the 6th International Wireless Communications and Mobile Computing Conference, in: IWCMC '10, ACM, New York, NY, USA, 2010, pp. 457–462, doi:10.1145/1815396.1815503.
[14] R. Braden, Requirements for internet hosts - communication layers, STD 3, RFC Editor, 1989. http://www.rfc-editor.org/rfc/rfc1122.txt (accessed 16.12.15).
[15] P. Wette, M. Dräxler, A. Schwabe, F. Wallaschek, M.H. Zahraee, H. Karl, MaxiNet: distributed emulation of software-defined networks, in: Proceedings of the IFIP Networking Conference, 2014.
[16] D. Belomestny, Rates of convergence for constrained deconvolution problem. preprint arxiv math.st/0306237 v1, 2003.
[17] S.L. Hakimi, On realizability of a set of integers as degrees of the vertices of a linear graph, J. Soc. Ind. Appl. Math. 10 (1962) 496–506.
[18] V. Havel, Poznámka o existenci konečných grafů, Časopis pro pěstování matematiky 080 (4) (1955) 477–480.
[19] P.L. Erdös, S.Z. Kiss, I. Miklós, L. Soukup, Constructing, sampling and counting graphical realizations of restricted degree sequences, arXiv math/1301.7523 v3 (2013).
[20] M. Mihail, N.K. Vishnoi, On generating graphs with prescribed vertex degrees for complex network modeling, in: Proceedings of the 3rd Workshop on Approximation and Randomization Algorithms in Communication Networks, 2002.
[21] M. Shreedhar, G. Varghese, Efficient fair queueing using deficit round robin, ACM SIGCOMM Comput. Commun. Rev. 25 (4) (1995) 231–242.
[22] D.H. Johnson, S. Sinanovic, et al., Symmetrizing the Kullback–Leibler Distance, IEEE Trans. Inf. Theory 1 (1) (2001) 1–10.
[23] D. Erickson, The beacon openflow controller, in: Proceedings of the Second Workshop on Hot Topics in Software Defined Networking (HotSDN), ACM, 2013.
[24] J.M. Chambers, W.S. Cleveland, B. Kleiner, P.A. Tukey, Graphical Methods for Data Analysis, Wadsworth, 1983.

Please cite this article as: P. Wette, H. Karl, DCT^{2}Gen: A traffic generator for data centers, Computer Communications (2015), http://dx.doi.org/10.1016/j.comcom.2015.12.001