TEACUP v0.4 – A System for Automated TCP Testbed Experiments

Sebastian Zander, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 140314A
Swinburne University of Technology
Melbourne, Australia
szander@swin.edu.au, garmitage@swin.edu.au

***Abstract*—Over the last few decades several TCP congestion control algorithms were developed in order to optimise TCP's behaviour in certain situations. While TCP was traditionally used mainly for file transfers, more recently it is also becoming the protocol of choice for streaming applications, for example YouTube or Netflix [1], [2]. Now there is even an ISO standard called Dynamic Adaptive Streaming over HTTP (DASH) [3]. However, the impact of different TCP congestion control algorithms on TCP-based streaming flows (within a mix of other typical traffic) is not well understood. Experiments in a controlled testbed allow us to shed more light on this issue. This report describes TEACUP (TCP Experiment Automation Controlled Using Python) version 0.4 – a software tool for running automated TCP experiments in a testbed. Based on a configuration file TEACUP can perform a series of experiments with different traffic mixes, different bottleneck configurations (such as bandwidths, queue mechanisms), different emulated network delay or loss, and different host settings (e.g. used TCP congestion control algorithm). For each experiment TEACUP automatically collects relevant information that allows analysing TCP behaviour, such as tcpdump files, SIFTR [4] and Web10G [5] logs.**

***Index Terms*—TCP, experiments, automated control**

I. Introduction

Over the last few decades several TCP congestion control algorithms were developed in order to optimise TCP's behaviour in certain situations. While TCP was traditionally used mainly for file transfers, more recently it is also becoming the protocol of choice for streaming applications, for example YouTube or Netflix [1], [2]. Now there is even an ISO standard called Dynamic Adaptive Streaming over HTTP (DASH) [3]. However, the impact of different TCP congestion control algorithms on TCP-based streaming flows (within a mix of other typical traffic) is not well understood. Experiments in a controlled testbed allow to shed more light on this issue.

This report describes TEACUP (TCP Experiment Automation Controlled Using Python) version 0.4 – a software tool for running automated TCP experiments in a testbed. Based on a configuration file TEACUP can perform a series of experiments with different traffic mixes, different bottlenecks (such as bandwidths, queue mechanisms), different emulated network delay or loss, and different host settings (e.g. TCP congestion control algorithm). For each experiment TEACUP automatically collects relevant information that allows analysing TCP behaviour, such as tcpdump files, SIFTR [4] and Web10G [5] logs. The related technical report [6] describes the design and implementation of the testbed.

This report is organised as follows. Section II provides some background information on Fabric. Section III describes the overall design of TEACUP. Section IV describes the configuration of TEACUP. Section V describes how to run experiments. Section VI describes how to analyse the data collected during experiments. Section VII describes utility functions that can be used for host maintenance. Section VIII outlines how to extend TEACUP. Section IX lists known issues. Section X concludes and outlines future work.

II. Fabric Background

TEACUP is build on the Python Fabric toolkit [7]. Here we provide a brief overview of Fabric and describe how to install it.

A. Overview

Fabric is a Python (2.5 or higher) library and command-line tool for streamlining the remote application deploy-

ment or system administration tasks using SSH [7]. Fabric provides several basic operations for executing local or remote shell commands and uploading/downloading files, as well as auxiliary functions, such as prompting the user for input, or aborting execution of the current task. Typically, with Fabric one creates a Python module where some functions are marked as Fabric tasks (using a Python function decorator).

These tasks can then be executed directly from the command line using the Fabric tool `fab`. The entry point of the module is a file commonly named `fabfile.py`, typically located in a directory from which we execute Fabric tasks (if the file is named differently we must use `fab -f <name>.py`). The complete list of tasks available in `fabfile.py` can be viewed with the command `fab -l`. Parameters can be passed to Fabric tasks, however a limitation is that all parameter values are passed as strings. Note that a Fabric task can execute another Fabric task with Fabric's execute() function.

Sections V, VI and VII contain a number of examples of how to run various TEACUP tasks.

B. Installation

TEACUP was developed with Fabric version 1.8, but it should run with newer versions of Fabric. The easiest way to install the latest version of Fabric is using the tool `pip`. Under FreeBSD `pip` can be installed with `portmaster`:

```
> portmaster devel/py-pip
```

On Linux `pip` can be installed with the package manager, for example on openSUSE it can be installed as follows:

```
> zypper install python-pip
```

Then to install Fabric execute:

```
> pip install fabric
```

You can test that Fabric is correctly installed:

```
> fab --version
Fabric 1.8.0
Paramiko 1.12.0
```

The Fabric manual provides more information about installing Fabric [8].

III. Design

This section describes the design of TEACUP. We first list the requirements. Then we describe the overall functional block design and the process flow. Next we describe the implemented traffic sources/sinks and loggers. Then we describe the host information logged. Then we describe the general host/router setup. Finally, we describe the naming scheme for log files.

A. Requirements

The following paragraphs describe the requirements for TEACUP.

1) General: Create a system to automate performing a series of TCP experiments with varying parameters.

2) Topology: The topology is a router with two testbed network interfaces (NICs) connected to two testbed subnets, with hosts on either side that act as traffic sources and sinks. Host could be moved between both test subnets if required, but this requires to re-cable the testbed.

3) TCP algorithms: The following TCP congestion control algorithms must be supported: NewReno and CUBIC (representing classic loss-based algorithms), CompoundTCP (Microsoft's hybrid), CDG (CAIA's hybrid), and HD (Hamilton Institutes' delay-based TCP. Optionally other TCPs may be supported. All the noted TCP algorithms are sender-side variants, so the destination can be any standard TCP implementation.

4) Path characteristics: Create bottleneck bandwidth limits to represent likely consumer experience (e.g. ADSL), and some data centre scenarios. Emulation of constant path delay and loss in either direction is required to simulate different conditions between traffic sources and sinks. The emulation is implemented by the bottleneck node (router).

5) Bottleneck AQM: The following Active Queuing Management (AQM) mechanisms are required: Tail-Drop/FIFO, CoDel, PIE, RED (somewhat deprecated, so less important given the wide range of potential RED parameter space). Optionally other AQM mechanisms may be supported. Since FreeBSD does not support some of the required AQMs the router has to be Linux. The buffer size must be configurable.

6) ECN Support: It must be possible to enable/disable Explicit Congestion Notification (ECN) (hosts and/or router).

7) Host OS: We are OS-agnostic. However, to cover the various TCP algorithms *and* their common implementations we will run scenarios where sources and/or destinations are Windows (Compound), Linux (NewReno, CUBIC) and FreeBSD (NewReno, CUBIC, CDG, HD)

8) Traffic loads: The following traffic loads must be supported: Streaming media over HTTP/TCP (DASH-like), TCP bulk transfer, UDP flows (VoIP-like), and data centre query/response patterns (one query to *N* responders, correlated return traffic causing incast congestion).

B. Overall design

The implementation of TEACUP is based on Fabric. It is designed based on multiple small tasks that are a) combined to run an experiment or a series of experiments but b) some may also be executed directly from the command line. Functions which are not tasks are ordinary Python functions. Currently, we do not make use of the object-oriented capabilities of Python.

Figure 1 shows the main functional building blocks. All the blocks in the diagram have corresponding Python files. However, we have summarised a number of Python files in the helper_functions block. The fabfile block is the entry point for the user. It implements tasks for running a single experiment or running a series of experiments with different parameters. The fabfile block also provides access to all other tasks via imports.

The experiment block implements the main function that controls a single experiment and uses a number of functions of other blocks. The sanity_checks block implements functions to check the config file, the presence of tools on the hosts, the connectivity between hosts, and a function to kill old processes still running. The host_setup block implements all functions to setup networking on the testbed hosts (this includes basic setup of the testbed router). The router_setup block implements the functions that setup the queues on the router and the delay/loss emulation. The loggers block implements the start and stop functions of the loggers, such as tcpdump and SIFTR/web10g loggers. The traffic_gens block implements the start and stop functions of all traffic generators.

The util block contains utility tasks that can be executed from the command line, such as executing a command on a number of testbed hosts or copying a file to a number of testbed hosts. The analysis block contains all the post-processing functions that extract measurement metrics from log files and plot graphs.

C. Experiment process flow

The following list explains the main steps that are executed during an experiment or a series of experiments.

I) Initialise and check config file
II) Get parameter combination for next experiment
III) Start experiment based on config and parameter configuration
    1) Log experiment test ID in file `experiments_started.txt`
    2) Get host information: OS, NIC names, NIC MAC addresses
    3) Reboot hosts: reboot hosts as required given the configuration
    4) Get host information again: OS, NIC names, NIC MAC addresses
    5) Run sanity checks
        - Check that tools to be used exist
        - Check connectivity between hosts
        - Kill any leftover processes on hosts
    6) Initialise hosts
        - Configure NICs (e.g. disable TSO)
        - Configure ECN use
        - Configure TCP congestion control
        - Initialise router queues
    7) Configure router queues: set router queues based on config
    8) Log host state: log host information (see Section III-F)
    9) Start all logging processes: tcpdump, SIFTR/Web10G etc.
    10) Start all traffic generators: start traffic generators based on config
    11) Wait for experiment to finish
    12) Stop all running processes on hosts
    13) Collect all log files from logging and traffic generating processes
    14) Log experiment test ID in file `experiments_completed.txt`
IV) If we have another parameter combination to run go to III, otherwise finish

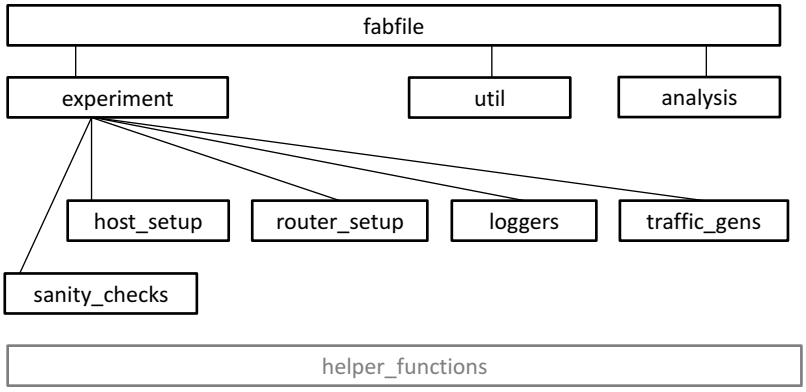

Figure 1. TEACUP main functional blocks

D. Traffic sources and sinks

We now describe the available traffic generator functions. How these can be used is described in more detail in Section IV-H.

1) iperf: The tool iperf [9] can be used to generate TCP bulk transfer flows. Note that the iperf client pushes data to the iperf server, so the data flows in the opposite direction compared to httperf. iperf can also be used to generate unidirectional UDP flows with a specified bandwidth and two iperfs can be combined to generate bidirectional UDP flows.

2) ping: This starts a ping from one host to another (ICMP Echo). The rate of pings is configurable for FreeBSD and Linux but limited to one ping per second for Windows.

3) lighttpd: A lighttpd [10] web server is started. This can be used as traffic source for httperf-based sinks. There are also automated script to setup fake content for DASH-like streaming and incast scenario traffic. However, for specific experiments one may need to setup web server content manually or create new scripts to do this. We choose lighttpd as web server because it is leaner than some other popular web servers and hence it is easier to configure and provides higher performance. Packages exist for FreeBSD, Linux and Cygwin.

4) httperf: The tool httperf [11] can be used to simulate an HTTP client. It can generate simple request patterns, such as accessing some .html file n times per second. It can also generate complex workloads based on work session log files (c.f. httperf man page at [11]).

5) httperf_dash: This starts a TCP video streaming httperf client [12] that emulates the behaviour of DASH

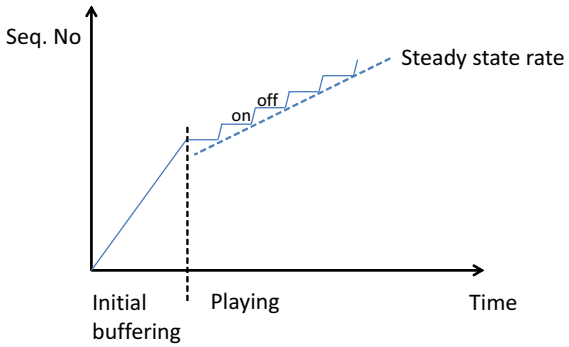

Figure 2. TCP video streaming (DASH) behaviour

or other similar TCP streaming algorithms [1], [2]. In the initial buffering phase the client will download the first part of the content as fast as possible. Then the client will fetch another block of content every t seconds. Figure 2 shows an illustration of this behaviour. The video rate and the cycle length are configurable (and the size of the data blocks depends on these).

6) httperf_incast: This starts an httperf client for the incast scenario. The client will request a block of content from n servers every t seconds. The requests are sent as close together as possible to make sure the servers respond simultaneously. The size of the data blocks is configurable. Emulating the incast problem in our physical testbed is difficult, since the number of hosts (number of responders) is relatively small.

7) nttcp: This starts an nttcp [13] client and an nttcp server for some simple unidirectional UDP VoIP flow emulation. The fixed packet size and inter-packet time can be configured. Note, nttcp also opens a TCP control connection between client and server. However, on this connection packets are only exchanged before and after

the data flow, and the number of control packets is relatively small.

E. Loggers

Currently, there are two types of loggers that log information on all hosts. All traffic is logged with tcpdump and TCP state information is logged with different tools.

1) Traffic logger: tcpdump is used to capture the traffic on all testbed NICs on all hosts. All traffic is captured, but the snap size is limited to 68 bytes by default.

2) TCP statistics logger: Different tools are used to log TCP state information on all hosts except the router. On FreeBSD we use SIFTR [4]. On Linux we use Web10G [5], which implements the TCP EStats MIB [14] inside the Linux kernel, and have implemented our own logging tool based on the Web10G library. On Windows 7 we implemented our own logging tool, which can access the TCP EStats MIB inside the Windows 7 kernel. The statistics collected by SIFTR are described in the SIFTR README [15]. The statistics collected by our Web10G client and the Windows 7 EStats logger are identical (based on the Web10G statistics) and described as part of the web100 (the predecessor of Web10G) documentation [16].

F. Host information logged

TEACUP does not only log the output of traffic generators and loggers, but also collects per-host information. This section describes the information collected for each host participating in an experiment. The following information is gathered *before* an experiment is started (<test_id> is some test ID):

- <test_id>_ifconfig.log.gz: This file contains the output of `ifconfig` (FreeBSD/Linux) or `ipconfig` (Windows).
- <test_id>_uname.log.gz: This file contains the output of `uname -a`.
- <test_id>_netstat.log.gz: This file contains information about routing obtained with `netstat -r`.
- <test_id>_ntp.log.gz: This file contains information about the NTP status based on `ntpq -p` (FreeBSD or Linux).
- <test_id>_procs.log.gz: This file contains the list of all running processes (output of `ps`).
- <test_id>_sysctl.log.gz: This file contains the output of `sysctl -a` (FreeBSD or Linux) and various information for Windows.
- <test_id>_config_vars.log.gz: This file contains information about all the V_ parameters in config.py (see Section IV). It logs the actual parameter values for each experiment. It also provides an indication of whether a variable was actually used or not (caveat: this does not work properly with variables used for TCP parameter configuration, they are always shown as used).
- <test_id>_host_tcp.log.gz: This file contains information of the TCP congestion control algorithm used on each host, and also any TCP parameter settings specified.

The following information is gathered *after* an experiment:

- <test_id>_queue_stats.log.gz: Information about the router queue setup (including all queue discipline parameters) and router queue and filtering statistics based on the output of `tc` (Linux) or `ipfw` (FreeBSD). Of course this information is collected *only* for the router.

G. Host setup

The setup of hosts other than the router is relatively straight-forward. First each host is booted into the selected OS. Then hardware offloading, such as TCP segmentation offloading (TSO) is disabled on testbed interfaces (all OS), the TCP host cache is disabled (Linux) or configured with a very short timeout and purged (FreeBSD), and TCP receive and send buffers are set to 2 MB or more (FreeBSD, Linux).

Next ECN is enabled/disable depending on the configuration. Then the TCP congestion control algorithm is configured for FreeBSD and Linux (including loading any necessary kernel modules). Then the parameters for the current TCP congestion control algorithm are configured if specified by the user (FreeBSD, Linux). Finally, custom user-specified commands are executed on hosts as specified in the configuration (these can overrule the general setup).

H. Linux router setup

The router setup differs between FreeBSD (where ipfw and Dummynet is used) and Linux (where tc and netem

is used). Our main focus is Linux, because Linux supports more AQM mechanisms than FreeBSD and some of the required AQM mechanisms are only implemented on Linux.

First, hardware offloading, such as TCP segmentation offloading (TSO) is disabled on the two testbed interfaces. Then the queuing is configured. Firstly, we describe our overall approach to setup rate limiting, AQM and delay/loss emulation for the Linux router. Secondly, we describe an example setup to illustrate the approach in practice.

1) Approach: We use the following approach. Shaping, AQM and delay/loss emulation is done on the egress NIC (as usual). The hierarchical token bucket (HTB) queuing discipline is used for rate limiting with the desired AQM queuing discipline (e.g. pfifo, codel) as leave node (this is similar to a setup mentioned at [17]). After rate shaping and AQM constant loss and delay is emulated with netem [18]. For each pipe we setup a new tc class on the two testbed NICs of the router. If pipes are unidirectional a class is only used on one of the two interfaces. Otherwise it is used on both interfaces. In future work we could optimise the unidirectional case and omit the creation of unused classes.

The traffic flow is as follows (also see Figure 8):

1) Arriving packets are marked by in the netfilter mangle table's POSTROUTING hook depending on source and destination IP address with a unique mark for each pipe.[1]
2) Marked packets are classified into the appropriate class based on the mark (a one-to-one mapping between marks and classes) and redirected to a pseudo interface. With pseudo device we refer to the so-called intermediate function block (IFB) device [19].
3) The traffic control rules on the pseudo interface do the shaping with HTB (bandwidth as per config) and the chosen AQM (as a leaf queuing discipline).
4) Packets go back to actual outgoing interface.
5) The traffic control rules on the actual interface do network delay/loss emulation with netem. We still need classes here to allow for pipe specific delay/loss settings. Hence we use a HTB again, but with the bandwidth set to the maximum possible rate (so there is effectively no rate shaping or AQM here) and netem+pfifo is used as leaf queuing discipline.[2]
6) Packets leave the router via stack / network card driver.

The main reason for this setup with pseudo interfaces is to cleanly separate the rate limiting and AQM from the netem delay/loss emulation. One could combine both on one interface, but then there are certain limitation, such as netem must be before the AQM and [17] reported that in such a setup netem causes problems. Also, a big advantage is that with our setup it is possible to emulate different delay/loss for different flows that share the same bottleneck/AQM.

2) Example: We now show an example of the setup based on partial (and for convenience reordered) output of a queue_stats.log.gz file for a scenario with two unidirectional pipes: 8 Mbps downstream and 1 Mbps upstream, both with 30 ms delay and 0% loss.

First, Figure 3 shows the netfilter marking rules. Our upstream direction is 172.16.10.0/24 to 172.16.11.0/24 and all packets are given the mark 0x1. Our downstream direction is 172.16.11.0/24 to 172.16.10.0/24 and all packets are given the mark 0x2.

In the upstream direction our outgoing interface is eth3 and we have the tc filters shown in Figure 4, which put each packet with mark 0x1 in class 1:1 and redirect it to pseudo interface ifb1.

Note that the class setting is effective for eth3, but it will not "stick" across interfaces. Hence we need to set the class again on ifb1 as shown in Figure 5 (again class 1:1 is set if the mark is 0x1):

On ifb1 we use the queuing discipline setup as shown in Figure 6. The HTB does the rate limiting to 1 Mbps. Here he leaf queuing discipline is a bfifo (byte FIFO) with a buffer size of 18.75 kB.

After packets are through the bfifo they are passed back to eth3 where we have an HTB with maximum rate and netem as leaf queuing discipline (here netem emulates 30 ms constant delay) as shown in Figure 7.

After leaving netem the packets are passed to the stack which passes the packets to the NIC driver. For the

[1] There also is a dummy rule "MARK and 0x0" inserted first, which is purely to count all packets going through the POSTROUTING hook. Note that since this dummy rule has 'anywhere' specified for source and destination, it also counts packets going through the router's control interface.

[2] The netem queue has a hard-coded size of 1000 packets, which should be large enough for our targeted experimental parameter space.

```
> iptables -t mangle -vL
Chain POSTROUTING (policy ACCEPT 52829 packets, 69M bytes) pkts bytes target prot opt in out
source destination
52988 69M MARK all -- any any anywhere anywhere MARK and 0x0
22774 1202K MARK all -- any any 172.16.10.0/24 172.16.11.0/24 MARK set 0x1
28936 66M MARK all -- any any 172.16.11.0/24 172.16.10.0/24 MARK set 0x2
```

Figure 3. Netfilter marking rules

```
> tc -s filter show dev eth3
filter parent 1: protocol ip pref 49152 fw
filter parent 1: protocol ip pref 49152 fw handle 0x1 classid 1:1
action order 33: mirred (Egress Redirect to device ifb1) stolen
index 3266 ref 1 bind 1 installed 99 sec used 12 sec
Action statistics:
Sent 1520865 bytes 22774 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
```

Figure 4. tc filter on outgoing network interface

```
> tc -d -s filter show dev ifb1
filter parent 1: protocol ip pref 49152 fw
filter parent 1: protocol ip pref 49152 fw handle 0x1 classid 1:1
```

Figure 5. tc filter on pseudo interface

```
> tc -d -s class show dev ifb1
class htb 1:1 root leaf 1001: prio 0 quantum 12500 rate 1000Kbit ceil 1000Kbit burst 1600b/1
mpu 0b overhead 0b cburst 1600b/1 mpu 0b overhead 0b level 0
Sent 1520865 bytes 22774 pkt (dropped 0, overlimits 0 requeues 0)
rate 62112bit 117pps backlog 0b 0p requeues 0
lended: 22774 borrowed: 0 giants: 0
tokens: 191750 ctokens: 191750
> tc -d -s qdisc show ifb1
qdisc htb 1: dev ifb1 root refcnt 2 r2q 10 default 0 direct_packets_stat 0 ver 3.17
Sent 1520865 bytes 22774 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
qdisc bfifo 1001: dev ifb1 parent 1:1 limit 18750b
Sent 1520865 bytes 22774 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
```

Figure 6. Queuing discipline setup on pseudo interface

sake of brevity we are not describing the downstream direction here, but the principle is exactly the same. The only differences are the interfaces used (eth2 and ifb0 instead of eth3 and ifb1) and the different HTB, AQM and netem parameters.

Figure 8 shows the flow of packets with the different steps carried out in the order of the numbers in parenthesis. The marking/classifying is not explicit, it takes place between step 1 and 2 (netfilter and class on actual interface) and between step 2 and 3 (class on pseudo interface).

We can see that with our setup it is possible to emulate different delay/loss for different flows that share the same bottleneck/AQM. Multiple tc filters on the ifb interface classify different flows as the same class so they share the same bottleneck. However, on the eth interface we have one class and one netem queue per flow and the tc filters classify each flow into a different class.

3) Notes: Note that in addition to the buffers mentioned earlier, according to [17] the HTB queuing discipline has a build-in buffer of one packet (cannot be changed) and the device drivers also have separate buffers.

I. FreeBSD router setup

While a Linux router is our main focus, we also implemented a basic router queue setup for FreeBSD.

On FreeBSD each pipe is realised as one Dummynet pipe, which does the rate shaping, loss/delay emulation

```
> tc -d -s class show dev eth3
class htb 1:1 root leaf 1001: prio 0 rate 1000Mbit ceil 1000Mbit burst 1375b cburst 1375b
Sent 1520865 bytes 22774 pkt (dropped 0, overlimits 0 requeues 0)
rate 62184bit 117pps backlog 0b 0p requeues 0
lended: 22774 borrowed: 0 giants: 0
tokens: 178 ctokens: 178
> tc -d -s qdisc show eth3
qdisc htb 1: dev eth3 root refcnt 9 r2q 10 default 0 direct_packets_stat 3 ver 3.17
Sent 1520991 bytes 22777 pkt (dropped 0, overlimits 66602 requeues 0)
backlog 0b 0p requeues 0
qdisc netem 1001: dev eth3 parent 1:1 limit 1000 delay 30.0ms
Sent 1520865 bytes 22774 pkt (dropped 0, overlimits 0 requeues 0)
backlog 0b 0p requeues 0
```

Figure 7. Queuing discipline setup on outgoing interface (netem)

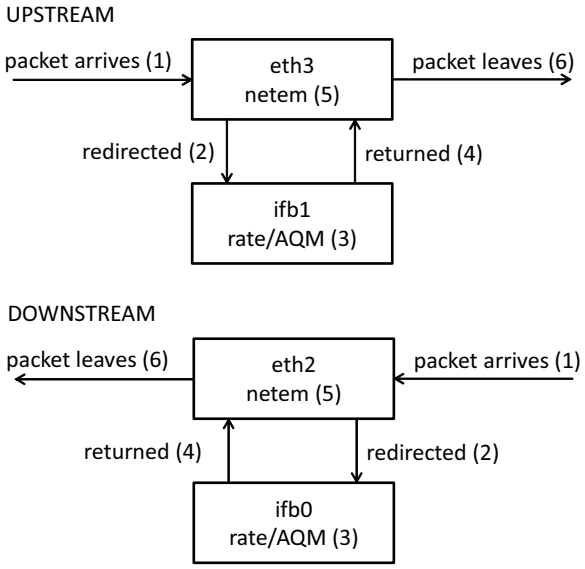

Figure 8. Flow of packets through our queue setup

and queuing (FIFO or RED only). ipfw rules are used to redirect packets to the pipes based on the specified source and destination IP parameters. If a pipe is unidirectional then there is a single "pipe X ip from <source> to <dest> out" rule. If a pipe is bidirectional there is an additional "pipe X ip from <dest> to <source> out" rule. A more sophisticated setup for FreeBSD remains future work.

J. Log file naming

The file names of TEACUP follow a naming scheme that has the following format:

`<test_ID_pfx>_[<par_name>_<par_val>_]*_<host>_[<traffgen_ID>_]_<file_name>.<extension>.gz`

The test ID prefix `<test_ID_pfx>` is the start of the file name and either specified in the config file (TPCONF_test_id) or on the command line (as described in Section V).

The `[<par_name>_<par_val>_]*` is the zero to n parameter names and parameter values (separated by an underscore). If an experiment was started with `run_experiment_single` there are zero parameter names and values. If an experiment was started with `run_experiment_multiple` there are as many parameters names and values as specified in TPCONF_vary_parameters. We also refer to the part `<test_ID_pfx>_[<par_name>_<par_val>_]*` (the part before the `<host>`) as test ID.

The `<host>` part specifies the IP or name of the testbed host a log file was collected from. This corresponds to an entry in TPCONF_router or TPCONF_hosts.

If the log file is from a traffic generator specified in TPCONF_traffic_gens, the traffic generator number follows the host identifier (`[<traffgen_ID>]`). Otherwise, `<traffgen_ID>` does not exist.

The `<file_name>` depends on the process which logged the data, for example it set to ‘uname’ for the uname information collected, it is set to ‘httperf_dash’ for an httperf client emulating DASH, or it set to ‘web10g’ for a Web10G log file. tcpdump files are somewhat special at the moment in that they have an empty file name for tcpdumps collected on hosts (assuming they only have one testbed NIC), or the file name is <int_name>_router for tcpdumps collected on the router (where <int_name> is name of the NIC, e.g. eth1).

The `<extension>` is either ‘dmp’ indicating a tcpdump file or ‘log’ for all other log files. All log files are usually gzip’d, hence their file names end with ‘.gz’.

Figure 9 shows an example name for a tcpdump file collected on host testhost2 for an experiment where two parameters (dash, tcp) where varied, and an example name for the output of one httperf traffic generator (traffic generator number 3) executed on host testhost2 for the same experiment.

IV. Config File

This section describes the TEACUP config.py file that controls the experiments.

A. V_variables

To iterate over parameter settings for each experiment TEACUP uses so-called V_variables. These are identifiers of the form V_<name>, where <name> must consist of only letters, numbers, hyphens (-) or underscores (_). V_variables can be used in router queue settings (see Section IV-F), traffic generator settings (see Section IV-H), TCP algorithm settings (see Section IV-I) or host setup commands (see Section IV-E). Section IV-K describes how to define V_variables.

B. Fabric configuration

The following settings in the config file are Fabric settings. For a more in-depth description refer to the Fabric documentation [7]. All Fabric settings are part of the Fabric env dictionary and hence Python variables (and must adhere to the Python syntax).

The user used for the SSH login is specified with env.user. For example:

```
env.user = 'root'
```

The password used for the SSH login is specified with env.password. The password can be empty if public-key authorisation is set up properly, e.g. the public SSH key of the control PC running TEACUP has been added to all hosts <user>/.ssh/authorized_keys files.

```
env.password = 'testroot'
```

The shell used to execute commands is specified with env.shell. By default Fabric uses Bash, but Bash is not standard on FreeBSD. So TEACUP's default setting is:

```
env.shell = '/bin/sh -c'
```

The timeout for an SSH connection is specified with env.timeout.

```
env.timeout = 5
```

The number of concurrent processes used for parallel execution is specified with env.pool_size. The number should be at least as high as the number of hosts, unless the number of hosts is large in which case we may want to limit the number of concurrent processes.

```
env.pool_size = 10
```

C. Testbed configuration

All TEACUP settings start with the TPCONF_ prefix and are Python variables (and must adhere to the Python syntax).

The path were the TEACUP scripts are located is specified with TPCONF_script_path. This is appended to the Python path.

```
TPCONF_script_path = '/home/test/src/teacup'
```

The path to the directory that is served by the TFTP server during the PXE boot process [6] is specified with TPCONF_tftpboot_dir. Setting this to an empty string means no PXE booting, and TPCONF_host_os and TPCONF_force_reboot (see below) are ignored.

```
TPCONF_tftpboot_dir = '/tftpboot'
```

Two lists specify the testbed hosts. TPCONF_router specifies the list of routers. Note that currently the TPCONF_router list is limited to only *one* router. TPCONF_hosts specifies the list of hosts. The hosts can be specified as IP addresses or host names (typically for convenience just the name without the domain part).

```
TPCONF_router = [ 'testhost1', ]
TPCONF_hosts = [ 'testhost2', 'testhost3' ]
```

The dictionary TPCONF_host_internal_ip specifies the testbed IP addresses for each host. The hosts (keys) specified must match the entries in the TPCONF_router and TPCONF_hosts lists exactly. The current code does simple string matching, it does *not* attempt to resolve host identifiers into some canonical form.

```
TPCONF_host_internal_ip = {
'testhost1' : ['172.16.10.1','172.16.11.1'],
'testhost2' : ['172.16.10.2'],
'testhost3' : ['172.16.10.3'],
}
```

```
# tcpdump file collected on testhost2 for an experiment where two parameters where varied
20131206-170846_windows_dash_1000_tcp_compound_testhost2.dmp.gz
# output of httperf traffic generator (traffic generator 3) executed on testhost2
20131206-170846_windows_dash_1000_tcp_compound_testhost2_3_httperf_dash.log.gz
```

Figure 9. Example file names

D. General experiment settings

TPCONF_test_id specified the default test ID prefix. Note that if the test ID prefix is specified on the command line, the command line overrules this setting.

```
now = datetime.datetime.today()
TPCONF_test_id = now.strftime("%Y%m%d-%H%M%S")
```

TPCONF_remote_dir specifies the directory on the remote host where the log files (e.g. tcpdump, SIFTR) are stored during an experiment. Files are deleted automatically at the end of an experiment, but if an experiment is interrupted files can remain. Currently, there is no automatic cleanup of the directory to remove left-over files.

```
TPCONF_remote_dir = '/tmp/'
```

The TPCONF_host_os dictionary specifies which OS are booted on the different hosts. The hosts (keys) specified must match the entries in the TPCONF_router and TPCONF_hosts lists exactly. The current code does simple string matching, it does *not* attempt to resolve host identifiers in some canonical form. The three different types of OS supported are: 'Linux', 'FreeBSD' and 'CYGWIN' (Windows). Selecting the specific Linux kernel to boot is not supported yet (the name of the kernel is hard-coded). The OS setting is valid for a whole series of experiment, or in other words the OS of hosts cannot be changed during a series of experiments.

```
TPCONF_host_os = {
'testhost1' : 'Linux',
'testhost2' : 'FreeBSD',
'testhost3' : 'Linux',
}
```

If TPCONF_force_reboot is set to '1' *all* hosts will be rebooted. If TPCONF_force_reboot is set to '0' only hosts where the currently running OS is *not* the desired OS (specified in TPCONF_host_os) will be rebooted.

```
TPCONF_force_reboot = '1'
```

TPCONF_boot_timeout specifies the maximum time in seconds (as integer) a reboot can take. If the rebooted machine is not up and running the chosen OS after this time, the reboot is deemed a failure and the script aborts, unless TPCONF_do_power_cycle is set to '1' (see below).

```
TPCONF_boot_timeout = 100
```

If TPCONF_do_power_cycle is set to '1' a host is power cycled if it does not reboot with TPCONF_boot_timeout seconds. If set to '0' there is no power cycling. If set to '1' the parameters TPCONF_host_power_ctrlport, TPCONF_power_admin_name and TPCONF_power_admin_pw must be set.

```
TPCONF_do_power_cycle = '0'
```

TPCONF_host_power_ctrlport is a dictionary that for each host specifies the IP (or host name) of the responsible power controller and the number of the power port the host is connected to (as integer starting from 1).

```
TPCONF_host_power_ctrlport = {
'testhost1' : ( '192.168.1.178', '1' ),
'testhost2' : ( '192.168.1.178', '2' ),
'testhost3' : ( '192.168.1.178', '3' ),
}
```

TPCONF_power_admin_name specifies the name of the power controller's admin user.

```
TPCONF_power_admin_name = 'admin'
```

TPCONF_power_admin_pw specifies the password of the power controller's admin user (which in the below example is identical to the SSH password used by Fabric, but in general it can be different).

```
TPCONF_power_admin_pw = env.password
```

It is possible to run custom per-host initialisation commands (see Section IV-E).

E. Custom host init commands

TPCONF_host_init_custom_cmds allows to execute custom per-host init commands. This allows to change the host configuration, for example with sysctls. TPCONF_host_init_custom_cmds is a dictionary, where

the key specifies the host name and the value is a list of commands. The commands are executed in exactly the order specified, after all default build-in host initialisation has been carried out. This means TPCONF_host_init_custom_cmds makes it possible to overrule default initialisation. The commands are specified as strings and are executed on the host exactly as specified (with the exception that V_variables, if present, are substituted with values). V_variables can be used in commands, but the current *limitation* is there can only be *one* V_variable per command.

Note that the commands are executed in the foreground, which means that for each command TEACUP will wait until it has been executed on the remote host before executing the next command for the *same* host. It is currently not possible to execute background commands. However, commands on different hosts are executed in parallel, i.e. waiting for a command to finish on host testhost1 does not block executing the next command on host testhost2. In summary, commands on different host are executed in parallel, but commands on the same host are executed sequentially.

The following config file part shows a simple example where we simply execute the command 'echo TEST' on host testhost1.

```
TPCONF_host_init_custom_cmds = {
'testhost1' : [ 'echo TEST', ],
}
```

F. Router queue setup

The variable TPCONF_router_queues specifies the router queues (Linux or FreeBSD). Each entry is a 2-tuple. The first value specifies a unique integer ID for each queue. The second value is a comma-separated string specifying the queue parameters. The queues do not necessarily need to be defined in the order of queue ID, but it is recommended to do so. The following queue parameters exist:

- **source**: Specifies the source IP/hostname or source network (<ip>[/<prefix>]) of traffic that is queued in this queue. If a host name is specified there can be no prefix. One can specify an internal/testbed or external/control IP/hostname. If an external IP/hostname is specified this will be automatically translated into the first internal IP specified for the host in TPCONF_host_internal_ip.
- **dest**: Specifies the destination IP/hostname or source network (<ip>[/<prefix>]) of traffic that is queued in this queue. If a host name is specified there can be no prefix. One can specify an internal/testbed or external/control IP/hostname. If an external IP/hostname is specified this will be automatically translated into the first internal IP specified for the host in TPCONF_host_internal_ip.
- **delay**: Specifies the emulated constant delay in milliseconds. For example, delay=50 sets the delay to 50 ms.
- **loss**: Specifies the emulated constant loss rate. For example, loss=0.01 sets the loss rate to 1%.
- **rate**: Specifies the rate limit of the queue. On Linux we can use units such as kbit or mbit. For example, queue_size='1mbit' sets the rate limit to 1 Mbit/second.
- **queue_size**: Specifies the size of the queue. On Linux queue size is usually defined in packets, only for some queuing disciplines we can specify the size in bytes. For example, if we have a Linux queue with size specified in packets, queue_size=1000 sets the queue size to 1000 packets. On FreeBSD the queue size is also specified in packets typically, but one can specify the size in bytes by adding a 'bytes' or 'kbytes', for example queue_size='100kbytes' specifies a queue of size 100 kbytes. If 'bdp' is specified the queue size will be set to the nominal bandwidth-delay-product (BDP) (this does only work for queuing disciplines where TEACUP knows whether the queue limit is in bytes or packets). The minimum queue size is one packet (if the size is specified in packets) or 2048 bytes (if the size is specified in bytes).
- **queue_size_mult**: The actual queue size is the queue size multiplied with this factor. This should only be used if queue_size if set to 'bdp'. This allows to vary the queue size in multiples of the nominal BDP.
- **queue_disc**: Specifies the queuing discipline. This can be the name of any of the queuing disciplines supported by Linux, such as 'fq_codel', 'codel', 'red', 'choke', 'pfifo', 'pie' etc. On FreeBSD the only queuing disciplines available are 'fifo' and 'red'. For example, queue_disc='fq_codel' sets the queuing discipline to the fair-queuing+codel model. For compatibility, with FreeBSD one can specify 'fifo' on Linux, which is mapped to 'pfifo' ('pfifo' is the default for HTB classes, which we use for

rate limiting). The queue_disc parameter must be specified explicitly.

- **queue_disc_params**: This parameter allows to pass parameters to queuing disciplines. For example, if we wanted to turn ECN on for fq_codel we would specify queue_disc_params='ecn' (c.f. fq_codel man page).
- **bidir**: This allows to specify whether a queue is unidirectional (set to '0') or bidirectional (set to '1'). A unidirectional queue will only get the traffic from source to dest, whereas a bidirectional queue will get the traffic from source to dest *and* from dest to source.
- **attach_to_queue**: This parameter works on Linux *only*. It allows to direct matching packets into an existing queue referenced by the specified queue ID, but to emulate flow-specific delay/loss (different from the delay and loss of other traffic). If attach_to_queue is specified the matching traffic will go through the already existing queue, but the emulated delay/loss is set according to the current queue specification. This means we can omit the rate, queue_disc and queue_size parameters, because they do not have any effect.
- **rtt**: This parameter allows to explicitly specify the emulated RTT in milliseconds. This parameter only needs to be specified if queue_size is set to 'bdp' and the RTT is not 2·delay of the current TEACUP queue (e.g. if we set up asymmetric delay with attach_to_queue).

All parameters must be assigned with either a constant value or a TEACUP variable or V_variable. V_variable names must be defined in TPCONF_parameter_list and TPCONF_variable_defaults (see below). V_variables are replaced with either the default value specified in TPCONF_variable_defaults or the current value from TPCONF_parameter_list if we iterate through multiple values for the parameter.

Figure 10 shows an example queue setup with the same delay/loss for every host and same delay/loss in both directions (all the parameters are variables here).

Figure 11 shows an example that illustrates the attach_to_queue parameter. Traffic between 172.16.10.3 and 172.16.11.3 goes through the same queues as traffic between 172.16.10.2 and 172.16.10.2, but in both directions it experiences twice the delay.

G. Traffic generator setup

Traffic generators are defined with the variable TPCONF_traffic_gens. This is a list of 3-tuples. The first value of a tuple is the start time relative to the start of the experiment. The second value of the tuple is the a unique ID. The third value of the tuple is a list of string containing the function name of the start function of the traffic generator as first entry followed by the parameters. The name of the functions and the parameters for each function are described in Section IV-H.

Client and server parameters can be external (control network) addresses or host names. An external address or host name is replaced by the first internal address specified for a host in TPCONF_host_internal_ip. Client and server parameters can also be internal (testbed network) addresses, which allows to specify any internal address.

Each parameter is defined as <parameter_name>=<parameter_value>. Parameter names must be the parameter names of traffic generator functions (and as such be valid Python variable names). Parameter values can be either constants (string or numeric) or V_variables that are replaced by the actual values depending on the current experiment. The V_variables must be defined in TPCONF_parameter_list and TPCONF_variable_defaults. Numeric V_variables can be modified using the normal mathematical operations, such as addition or multiplication. For example, if a variable V_delay exists one can specify 2·V_delay as parameter value.

Figure 12 shows a simple example. At time zero a web server is started and fake DASH content is created. 0.5 seconds later a httperf DASH-like client is started. The duration and rate of the DASH-like flow are specified by variables that can change for each experiment. In contrast the cycle length and prefetch time are set to fixed values.

The example config.py file in the source code distribution contains more examples on setting up traffic generators.

H. Available traffic generators

This section describes the traffic generators (listed by their start function names) that can be used in TPCONF_traffic_gens.

```
TPCONF_router_queues = [
( '1', "source='172.16.10.0/24', dest='172.16.11.0/24', delay=V_delay, loss=V_loss, rate=V_urate,
       queue_disc=V_aqm, queue_size=V_bsize" ),
( '2', "source='172.16.11.0/24', dest='172.16.10.0/24', delay=V_delay, loss=V_loss, rate=V_drate,
       queue_disc=V_aqm, queue_size=V_bsize" ),
]
```

Figure 10. Router queue definition example

```
TPCONF_router_queues = [
( '1', "source='172.16.10.2', dest='172.16.11.2', delay=V_delay, loss=V_loss, rate=V_up_rate,
       queue_disc=V_aqm, queue_size=V_bsize" ),
( '2', "source='172.16.11.2', dest='172.16.10.2', delay=V_delay, loss=V_loss, rate=V_down_rate,
       queue_disc=V_aqm, queue_size=V_bsize" ),
( '3', "source='172.16.10.3', dest='172.16.11.3', delay=2*V_delay, loss=V_loss, attach_to_queue='1'" ),
( '4', "source='172.16.11.3', dest='172.16.10.3', delay=2*V_delay, loss=V_loss, attach_to_queue='2'" ),
]
```

Figure 11. Router queue definition with attach_to_queue example

```
TPCONF_traffic_gens = [
( '0.0', '1', "start_http_server, server='testhost3', port=80" ),
( '0.0', '2', "create_http_dash_content, server='testhost3', duration=2*V_duration,
               rates=V_dash_rates, cycles='5, 10'" ),
( '0.5', '3', "start_httperf_dash, client='testhost2', server='testhost3', port=80,
               duration=V_duration, rate=V_dash_rate, cycle=5, prefetch=2" ),
]
```

Figure 12. Traffic generator example

1) start_iperf: This starts an iperf client and server. Note that the client sends data to the server. It has the following parameters:

- **port**: port number to use for client and server (passed to iperf -p option)
- **client**: IP or name of client (passed to iperf -c option)
- **server**: IP or name of server (passed to iperf -B option)
- **duration**: time in seconds the client transmits (passed to iperf -t option)
- **congestion_algo**: TCP congestion algorithm to use (works only on Linux)
- **mss**: TCP maximum segment size (passed to iperf -M option)
- **buf_size**: Send and receive buffer size in bytes (passed to -j and -k, iperf with CAIA patch [6])
- **proto**: Protocol to use, 'tcp' (default) or 'udp' (sets iperf -u option for 'udp')
- **rate**: The bandwidth used for TCP (passed to iperf -a option) or UDP (passed to iperf -b option). Can end in 'K' or 'M' to indicate kilo bytes or mega bytes.
- **extra_params_client**: Command line parameters passed to iperf client
- **extra_params_server**: Command line parameters passed to iperf server

2) start_ping: This starts a ping and has the following parameters:

- **client**: IP or name of machine to run ping on
- **dest**: IP or name of machine to ping
- **rate**: number of pings per second (Windows 7 ping only supports 1 ping/second) (default = 1)
- **extra_params**: Command line parameters passed to ping

3) start_http_server: This starts an HTTP server (lighttpd) and has the following parameters:

- **port**: port to listen on (currently one server can listen on only one port)
- **config_dir**: directory where the config file (lighttpd.conf) should be copied to
- **config_in**: local template for config file
- **docroot**: document root on server (FreeBSD default: /usr/local/www/data, Linux/CYGWIN default: /srv/www/htdocs)

4) create_http_dash_content: This creates fake content for the DASH-like client. It has the following parameters:

- **server**: IP or name of HTTP server
- **docroot**: document root of HTTP server (FreeBSD default: /usr/local/www/data, Linux/CYGWIN default: /srv/www/htdocs)
- **duration**: number of seconds of fake content
- **rates**: comma-separated list of DASH rates in kB
- **cycles**: comma-separated list of cycle length in seconds

5) create_http_incast_content: This creates fake content for incast experiments. It has the following parameters:

- **server**: IP or name of HTTP server
- **docroot**: document root of HTTP server (FreeBSD default: /usr/local/www/data, Linux/CYGWIN default: /srv/www/htdocs)
- **sizes**: comma-separated list of content file sizes

6) start_httperf: This starts an httperf HTTP client. It has the following parameters:

- **client**: IP or name of client
- **server**: IP or name of HTTP server (passed to httperf --server)
- **port**: port server is listening on (passed to httperf --port)
- **conns**: total number of connections (passed to httperf --num-conns)
- **rate**: rate at which connections are created (passed to httperf --rate)
- **timeout**: timeout for each connection; httperf will give up if a HTTP request does not complete within the timeout (passed to httperf --timeout)
- **calls**: number of requests in each connection (passed to httperf --num-calls, default = 1)
- **burst**: length of burst (passed to httperf --burst-length)
- **wsesslog**: session description file (passed to third parameter of httperf --wsesslog)
- **wsesslog_timeout**: default timeout for each wsesslog connection (passsed to second parameter of httperf --wsesslog)
- **period**: time between creation of connections; equivalent to 1/rate if period is a number, but period can also specify inter-arrival time distributions (see httperf man page)
- **sessions**: number of sessions (passed to first parameter of httperf --wsesslog, default = 1)
- **call_stats**: number of entries in call statistics array (passed to httperf --call-stats, default = 1000)
- **extra_params**: Command line parameters passed to httperf

7) start_httperf_dash: This starts a DASH-like httperf client. It has the following parameters:

- **client**: IP or name of client
- **server**: IP or name of HTTP server (passed to httperf --server)
- **port**: port server is listening on (passed to httperf --port)
- **duration**: duration of DASH flow in seconds
- **rate**: data rate of DASH-like flow in kbps
- **cycle**: interval between requests in seconds
- **prefetch**: prefetch time in seconds of content to prefetch (can be fractional number) (default = 0.0)
- **prefetch_timeout**: like timeout for start_httperf but only for the prefetch (by default this is set to cycle time)
- **extra_params**: Command line parameters passed to httperf

The behaviour is as follows:

1) The client opens a persistent TCP connection to the server.
2) If prefetch is > 0.0 the client will fetch the specified number of seconds of content and right after that send a request for the next block (step 3).
3) The client will request a block of content, wait for some time (cycle minus download time) and then request the next block. The size of one block is $\text{cycle} \cdot \text{rate} \cdot 1000/8$ bytes.

8) start_httperf_incast: This starts an httperf client for the incast scenario. It has the following parameters:

- **client**: IP or name of client
- **server**: IP or name of HTTP server (passed to httperf --server)
- **port**: port server is listening on (passed to httperf --port)
- **period**: period between requests in seconds (floating point number)
- **burst_size**: number of queries sent at each period start (this is only to increase the number of queries in our testbed, which only has a few responders)

- **response_size**: size of response from each responder in kB
- **extra_params**: Command line parameters passed to httperf

9) start_nttcp: This starts an nttcp client and an nttcp server for some simple unidirectional UDP VoIP flow emulation. It has the following parameters:

- **client**: IP or name of client
- **server**: IP or name of HTTP server
- **port**: control port server is listening on (passed to nttcp -p)
- **duration**: duration of the flow (based on this and interval TEACUP computes the number of buffer to send, passed to nttcp -n)
- **interval**: interval between UDP packets in milliseconds (passed to nttcp -g)
- **psize**: UDP payload size (excluding UDP/IP header) (passed to nttcp -l)
- **buf_size**: send buffer size (passed to nttcp -w)
- **extra_params_client**: Command line parameters passed to nttcp client
- **extra_params_server**: Command line parameters passed to nttcp server

Note that nttcp also opens a TCP control connection between client and server. However, on this connection packets are only exchanged before and after the data flow, and the number of exchanged TCP packet is very small.

I. Mandatory experiment variables

We now describe the mandatory experiment variables. These must be in every config file. There are two types of variables: singulars and lists. Singulars are fixed parameters while lists specify the different values used in subsequent experiments based on the definitions of TPCONF_parameter_list and TPCONF_vary_parameters (see below).

The duration of the experiment in seconds (must be an integer) is specified with TPCONF_duration. Note that currently the actual duration of an experiment is the number of seconds specified by TPCONF_duration *plus* the number of seconds until the last traffic generator is started (based on TPCONF_traffic_gens). TPCONF_duration is specified as follows:

```
TPCONF_duration = 30
```

The number of runs carried out for each unique parameter combination is specified with TPCONF_runs:

```
TPCONF_runs = 1
```

TPCONF_ECN specifies whether ECN is used. If set to '1' ECN is enabled for all hosts. If set to '0' ECN is disabled for all hosts. Currently per-host configuration is not possible. Note that this only turns on ECN on the hosts, but the AQM mechanism on the router must also be configured to use ECN.

```
TPCONF_ECN = [ '0', '1' ]
```

TPCONF_TCP_algos specifies the TCP congestion algorithms used. The following algorithms can be selected: 'newreno', 'cubic', 'hd', 'htcp', 'cdg', 'compound'. However, only some of these are supported depending on the OS a host is running:

- Windows: compound (default) only;
- FreeBSD: newreno (default), cubic, hd, htcp, cdg;
- Linux: newreno, cubic (default), htcp.

```
TPCONF_TCP_algos = [ 'newreno', 'cubic', ]
```

Instead of specifying a particular TCP algorithm one can specify 'default'. This will set the algorithm to the default algorithm depending on the OS the host is running. Using only TPCONF_TCP_algos one is limited to either using the same algorithm on all hosts or the defaults. To run different algorithms on different hosts, one can specify 'host_<N>' where <N> is an integer number starting from 0. The <N> refers to the Nth entry for each host in TPCONF_host_TCP_algos (see below).

TPCONF_host_TCP_algos defines the TCP congestion control algorithms used for each host if the 'host_<N>' definitions are used in TPCONF_TCP_algos. In the following example a 'host_0' entry in TPCONF_TCP_algos will lead to each host using its default. A 'host_1' entry will set testhost2 to use 'newreno' and testhost3 to use 'cdg'.

```
TPCONF_host_TCP_algos = {
'testhost2' : [ 'default', 'newreno', ],
'testhost3' : [ 'default', 'cdg', ],
}
```

With TPCONF_host_TCP_algo_params we can specify parameter settings for each host and TCP congestion control algorithm. The settings are passed directly to sysctl on the remote host. We can use V_variables to iterate over different settings (similar as for pipes and

traffic generators) and these are replaced with the actual current value before passing the string to sysctl. For example, we can specify settings for CDG for host testhost2:

```
TPCONF_host_TCP_algo_params = {
'testhost2' : { 'cdg' : [
'net.inet.tcp.cc.cdg.beta_delay = V_cdgbdel',
'net.inet.tcp.cc.cdg.beta_loss = V_cdgbloss',
'net.inet.tcp.cc.cdg.exp_backoff_scale = 3',
'net.inet.tcp.cc.cdg.smoothing_factor = 8' ]
}}
```

TPCONF_delays specifies the emulated delays in milliseconds. The numbers must be chosen from [0, max_delay). For most practical purposes the maximum delay max_delay is 10 seconds, although it could be more (if supported by the emulator). The following shows an example:

```
TPCONF_delays = [ 0, 25, 50, 100 ]
```

TPCONF_loss_rates specifies the emulated loss rates. The numbers must be between 0.0 and 1.0. The following shows an example:

```
TPCONF_loss_rates = [ 0, 0.001, 0.01 ]
```

TPCONF_bandwidths specifies the emulated bandwidths as 2-tuples. The first value in each tuple is the downstream rate and the second value in each tuple is the upstream rate. Note that the values are passed through to the router queues and are not checked by TEACUP. Units can be used if the queue setup allows this, e.g. in the following example we use mbit to specify Mbit/second which Linux tc allows us to do:

```
TPCONF_bandwidths = [
( '8mbit', '1mbit' ),
( '20mbit', '1.4mbit' ),
]
```

TPCONF_aqms specifies the list of AQM/queuing techniques. This is completely dependent on the router OS. Linux supports 'fifo' (mapped to 'pfifo'), 'pfifo', 'bfifo', 'fq_codel', 'codel', 'pie', 'red' etc. (refer to the tc man page for the full list). FreeBSD support only 'fifo' and 'red'. Default on Linux and FreeBSD are FIFOs (with size in packets). The following shows an example:

```
TPCONF_aqms = [ 'pfifo', 'fq_codel', 'pie' ]
```

TPCONF_buffer_sizes specifies the bottleneck buffer sizes. If the router is Linux this is mostly in packets/slots, but it depends on the AQM technique (e.g. for bfifo it is bytes). If the router is FreeBSD this would be in slots by default, but we can specify byte sizes (e.g. we can specify 4Kbytes). The following shows an example:

```
TPCONF_buffer_sizes = [ 1000, 1 ]
```

TPCONF_vary_parameters specifies this list of parameters we vary, i.e. parameters that have multiple values. These parameters must be defined in TPCONF_parameter_list (see below). The total number of experiments carried out is the number of unique parameter combinations multiplied by the number of TPCONF_runs if 'runs' is specified here. The TPCONF_vary_parameters specification also defines the order of the parameters in the file names. While not strictly necessary 'runs' should be last in the list (if used). If 'runs' is not specified, there is a single experiment for each parameter combination. TPCONF_vary_parameters is only used for multiple experiments. When we run a single experiment (run_experiment_single) all the variables are set to fixed values based on TPCONF_variable_defaults. The following shows an example for the parameters included TPCONF_parameter_list in the example config.py:

```
TPCONF_vary_parameters = [
        'tcpalgos', 'delays', 'loss',
        'bandwidths', 'aqms', 'bsizes',
        'runs',
]
```

J. Experiment-specific variables

Some variables defined in the example config.py file are only used with certain traffic generators.

TPCONF_dash_rates specifies the DASH content rates in kbit/second and TPCONF_dash_rates_str is a string with a comma-separated list of rates (the latter is used by the content creation function create_http_dash_content). DASH rates must be integers. The following shows an example:

```
TPCONF_dash_rates = [ 500, 1000, 2000 ]
TPCONF_dash_rates_str = ','.join(map(str,
                TPCONF_dash_rates))
```

TPCONF_inc_content_sizes specifies the content sizes in kB for the replies send in an incast scenario (as integer) and TPCONF_inc_periods specifies the length of a period between requests by the querier in full seconds (as floating point). The following shows an example:

```
TPCONF_inc_content_sizes= '64, 512, 1024'
TPCONF_inc_periods = [ 10 ]
```

K. Defining variables

TPCONF_parameter_list specifies the variable parameters. Note that despite its name TPCONF_parameter_list is a Python dictionary. The keys are names that can be used in TPCONF_vary_parameters. The values are 4-tuples. The first parameter of each tuple is a list of V_variables that can be used, for example in the queue configuration or traffic generator configuration. The second parameter of each tuple is a list of names used in the file names of created log files. The third parameter of each tuple is the list of parameter values, these are usually references to the lists defined in the previous section. The last parameter is a dictionary of extra V_variables (this can be an empty dictionary). The keys are variable names and the values are the variable values.

The length of the first three tuple parameters (V_variable identifiers, short names and V_variable values) must be equal. When a series of experiments is started with 'fab run_experiment_multiple' the following happens. For each parameter combination of the vary parameter defined in TPCONF_vary_parameters one experiment is run where the parameter settings are logged in the file name using the short names, and the V_ variables are set to the parameter combination given by the value lists.

TPCONF_parameter_list can handle grouped V_variables, such as TPCONF_bandwidths, where in each experiment a specific combination of the grouped V_variables is specified.

TPCONF_variable_defaults is a dictionary that specifies the defaults for V_ variables. The keys are V_ variable names and the values are the default values (often the first value of the parameter lists). For each parameter that is not varied, the default value specified in TPCONF_variable_defaults is used.

We now discuss a simple example where we focus on the variables to vary delay and TCP algorithms. Assume we want to experiment with two delay settings and two different TCP CC algorithms. So we have:

```
TPCONF_delays = [ 0, 50 ]
TPCONF_TCP_algos = [ 'newreno', 'cubic' ]
```

We also need to specify the two parameters to be varied and the default parameters for the variables as shown in Figure 13.

V_delay can be used in the router queue settings. V_tcp_cc_algo is passed to the host setup function. When we run 'fab run_experiment_multiple' this will run the following experiments, here represented by the start of the log file names also called the test ID (we assume the test ID prefix is the default from Section IV-D):

```
20131206-170846_del_0_tcp_newreno
20131206-170846_del_0_tcp_cubic
20131206-170846_del_50_tcp_newreno
20131206-170846_del_50_tcp_cubic
```

L. Adding new V_ variables

New V_variables are easy to add. Say we want to create a new V_variable called V_varx. We need to do the following:

1) Add a new list with parameter values, let's say TPCONF_varx = [x,y]
2) Add one line in TPCONF_parameters_list: the key is the identifier that can be used in TPCONF_vary_parameters (let's call it varx_vals), and the value is a 4-tuple: variable name as string 'V_varx', variable name used in file name (say 'varx'), pointer to value list (here TPCONF_varx), and optionally a dictionary with related variables (here empty).
3) Add one line in TPCONF_variable_defaults specifying the default value used (when not iterating over the variable): the key is the V_variable name as string (here 'V_varx') and the value is the default value from TPCONF_varx (say TPCONF_varx[0]).

Technically, step 1 is not necessary as the the list of values can be put directly in TPCONF_parameters_list and the default value can be put directly in TPCONF_variable_defaults. However, defining a separate list improves the readability.

V. Running Experiments

This section describes how to run experiments. First, we describe the initial steps needed. Then we outline a simple example config file. Finally, we describe how to run the experiments.

```
TPCONF_parameter_list = {
'delays'   : ( [ 'V_delay' ],       [ 'del' ], TPCONF_delays,    {} ),
'tcpalgos' : ( [ 'V_tcp_cc_algo' ], [ 'tcp' ], TPCONF_TCP_algos, {} ),
}
TPCONF_variable_defaults {
'V_delay'    : TPCONF_delays[0]',
'V_tcp_algo' : TPCONF_TCP_algos[0],
}
TPCONF_vary_parameters = [ 'delays', 'tcpalgos' ]
```

Figure 13. Specifying the parameters to be varied

A. Initial steps

First you should create a new directory for the experiment or series of experiments. Copy the files fabfile.py and run.sh from the TEACUP distribution into that new directory. Then create a config.py file in the directory. An easy way to get a config.py file is to start with the provided example config.py as basis and modify it as necessary.

B. Example config

Figure 14 shows a minimal but complete config.py file. Our testbed consists of three machines here, two hosts (192.168.1.2, 192.168.1.3) connected by a router (192.168.1.4). The two hosts will run FreeBSD for the experiment, while the router will run Linux. On the router we configure two pipes, one in each direction, with different rates but the same AQM mechanism, buffer size, and emulated delay and loss. The test traffic consists of two parallel TCP sessions generated with iperf, both start at the start of the experiment (time 0.0). With iperf the client sends data to the server, so the data is send from 192.168.1.2 to 192.168.1.3. Each experiment lasts 30 seconds and we run a series of experiments varying the TCP congestion control algorithm, the network delay and loss, the upstream and downstream bandwidths, the AQM technique, and the buffer size. There is one experiment for each parameter combinations (one run).

C. Running experiments

There are two tasks to start experiments: run_experiment_single and run_experiment_multiple.

To run a single experiment with the default test ID prefix TPCONF_test_id, type:

```
> fab run_experiment_single
```

To run a series of experiment based on the TPCONF_vary_parameters setting with the default test ID prefix TPCONF_test_id, type:

```
> fab run_experiment_multiple
```

In both cases the Fabric log output will be printed out on the current terminal and it can be redirected with the usual means. The default test ID prefix TPCONF_test_id is specified in the config file. However, the test ID prefix can also be specified on the command line (overruling the config setting):

```
> fab run_experiment_multiple:test_id=`date
+"%Y%m%d-%H%M%S"`
```

The last command will run a series of experiments where the test ID prefix is YYYYMMDD-HHMMSS, using the actual date when the fab command is run. For convenience a shell script `run.sh` shell exists. The shell script logs the Fabric output in a <test_ID_prefix>.log file and is started with:

```
> run.sh
```

The shell script generates a test ID prefix and then executes the command:

```
> fab run_experiment_multiple:test_id=
<test_ID_pfx> > <test_ID_pfx>.log 2>&1
```

The test ID prefix is set to `` `date +"%Y%m%d-%H%M%S"`_experiment ``. The output is unbuffered, so one can use `tail -f` on the log file and get timely output. The fabfile to be used can be specified, i.e. to use the fabfile `myfabfile.py` instead of `fabfile.py` run:

```
> run.sh myfabfile.py
```

The run_experiment_single and run_experiment_multiple tasks keeps track of experiments using two files in the current directory:

- The file `experiment_started.txt` logs the test IDs of all experiments started.
- The file `experiment_completed.txt` logs the test IDs of all experiments *successfully completed*.

Note that both of these files are never reset by TEACUP. New test IDs are simply appended to the current files if they already exist. It is the user's responsibility to delete the files in order to reset the list of experiments.

It is possible to resume an interrupted series of experiments started with run_experiment_multiple with the resume parameter. All experiments of the series that were not completed (not logged in `experiments_completed.txt`) are done again. For example, the following command resumes a series of experiments with test ID prefix `20131218-113431_windows` (and appends the log output to the existing log file):

```
> fab run_experiment_multiple:
test_id=20131218-113431_windows,resume=1
>> 20131218-113431_windows.log 2>&1
```

The `resume` parameter makes it possible to redo selected experiments, even if they were successfully completed, by removing them from `experiments_completed.txt` (using a text editor).

VI. Analysing Experiment Data

This section describes how to analyse the data of an experiment or a series of experiments. First, we describe the available functions to extract data and plot graphs. Then we describe shell scripts that can be used to combine multiple graphs on the same page to allow easy comparison of the results of different experiments.

A. Analysis functions

Currently analysis functions exist for:

1) Plotting the throughput including all header bytes (based on tcpdump data);
2) Plotting the Round Trip Time (RTT) using SPP [20], [21] (based on tcpdump data);
3) Plotting the TCP congestion window size (CWND) (based on SIFTR and Web10G data);
4) Plotting the TCP RTT estimate (based on SIFTR and Web10G data). The function can plot both, the smoothed estimate and an unsmoothed estimate (also for SIFTR the unsmoothed estimate is the improved ERTT [22] estimate);
5) Plotting an arbitrary TCP statistics from SIFTR and Web10G data.

A convenience function exists that plots graphs 1–4 (listed above). The easiest way to generate all graphs for all experiments is to run the following command in the directory containing the experiment data:

```
> fab analyse_all
```

This command will generate results for all experiments listed in the file `experiments_completed.txt`. By default the TCP RTT graphs generated are for the smoothed RTT estimates and in case of SIFTR this is not the ERTT estimates (if the smoothed parameter is set to '0', non-smoothed estimates are plotted and in the case of SIFTR this is the ERTT estimates). The analysis can be run for only single experiment by specifying a test ID. The following command generates all graphs for the experiment `20131206-102931_dash_2000_tcp_newreno`:

```
> fab analyse_all:test_id=
20131206-102931_dash_2000_tcp_newreno
```

To generate a particular graph for a particular experiment one can use the specific analysis function (analyse_throughput, analyse_spp_rtt, analyse_cwnd, analyse_tcp_rtt) together with a test ID (specifying the test ID is mandatory in this case). For example, the following command only generates the TCP RTT graph for the non-smoothed estimates:

```
> fab analyse_tcp_rtt:test_id=
20131206-102931_dash_2000_tcp_newreno,
smoothed=0
```

Note, the smoothed parameter can also be used with analyse_all. The following command only generates the throughput graph:

```
> fab analyse_throughput:test_id=
20131206-102931_dash_2000_tcp_newreno
```

The `analyse_tcp_stat` function can be used to plot any TCP statistic from SIFTR or Web10G logs. For example, we can plot the number of kilo bytes in the send buffer at any given time with the command:

```
> fab analyse_tcp_stat:test_id=
20131206-102931_tcp_newreno,out_dir=./results,
siftr_index=22,web10g_index=116, ylabel="Snd
buf (kbytes)",yscaler=0.001
```

The siftr_index defines the index of the column of the statistic to plot for SIFTR log files. The web10g_index defines the index of the column of the statistic to plot for Web10G log files. If one has only SIFTR or only Web10G log files the other index does not need to be specified. But for experiments with SIFTR *and* Web10G log files both indexes must be specified. By default both indexes are set to plot CWND for SIFTR and Web10G logs. The lists of available statistics (including the column numbers) are in the SIFTR README [15] and the Web10G documentation [16].

The analyse_dash_goodput task allows to plot the goodput for DASH-like traffic over time. The plot is based on data from the httperf log files of the DASH-like clients (named <test_id>_httperf_dash.log.gz). The task can be used as follows:

```
> fab analyse_dash_goodput:
test_id=20131218-182744,
dash_log_list=dash_logs.txt,out_dir=./results/,
lnames="newreno;cdg;vegas"
```

The analyse_dash_goodput task has the common test_id, out_dir and plot_only parameters. By default the task extracts data from all client log files that have the test ID specified. However, if dash_log_list is specified the task extracts the data from the log files listed in a text file (the value of dash_log_list is the file name). This allows to control exactly which dash client logs are used, and these files can be from experiments with different test IDs. The format of the log list file is one file name per line.

By default the legend names are the file names (minus the '_httperf_dash.log.gz' part). The parameter lnames allows to specify the legend names used. The number of legend names specified must be equal to the number of files names specified in the log list file (if dash_log_list is used) or equal to the number of log files with the specified test ID (if dash_log_list is not used).

Note currently the underlying plot function for analyse_throughput, analyse_spp_rtt, analyse_cwnd, analyse_tcp_rtt, analyse_tcp_stat can only plot 12 different time series on a single graph. If the number of data series to plot is larger than 12, multiple graphs are generated with a _<graph_number> at the end of each file name to indicate the number of the graph in the series of graphs (graph number starting from 1).

B. Analysis functions options

All analysis functions have a parameter `replot_only`. This parameter allows to replot the graphs without extracting the data again (from tcpdump, SIFTR, Web10G files). For example, the following command recreates the graphs without extracting the data again:

```
> fab analyse_all:replot_only=1
```

By default all result files are generated in the directory where fab is executed. To put the output files into a specific directory the `out_dir` parameter can be specified:

```
> fab analyse_all:out_dir=./results/
```

This will put all output files into a sub-directory of the current directory named results (which must be created prior to executing the command). Of course you can also specify absolute paths.

In many experiments we may have TCP flows where data only/mostly flows in one direction and TCP statistics in the other direction are basically constant). The omit_const parameter can be used to suppress any completely constant series (i.e. all values are identical). It can be used as follows:

```
> fab analyse_all:omit_const=1
```

Any flows that have only very few data points (less or equal than `min_values`) are excluded from the plot (by default min_values = 3). The min_values parameter can be changed on the command line, for example the following command omits any flows with 20 data points or less from the plots:

```
> fab analyse_all:min_values=20
```

If an analysis_all was interrupted (e.g. because a log file was corrupted) we can resume the analysis after the experiment with the corrupted files. First, one needs to look up the next test ID after the corrupted test ID in `experiments_completed.txt`. Then, one can resume at this test ID using the resume_id parameter. For example, if for a test ID 20131206-102931_dash_2000_tcp_newreno_run_0
we cannot do the analysis because of corrupted data files and the next test ID is 20131206-102931_dash_2000_tcp_newreno_run1, we can continue the analysis with this command:

```
> fab analyse_all:resume_id=
20131206-102931_dash_2000_tcp_newreno_run_1
```

For analyse_all the parameter exp_list allows to change the file used as list of test IDs (by default `experiments_completed.txt`), which makes it possible to adjust the list of experiments we generate results for. The following shows an example:

```
> fab analyse_all:exp_list=myexp_list.txt,
out_dir="./results"
```

The tasks analyse_throughput, analyse_spp_rtt, analyse_cwnd, analyse_tcp_rtt and analyse_tcp_stat have a parameter ymax. This parameter can be used to set the y-axis limit to a specific value in order to produce multiple plots with the same scale (by default the y-axis limit is determined automatically). It can be used as follows (here the y-axis limit is set to 200 ms):

```
> fab analyse_spp_rtt:test_id=
20131206-102931_dash_2000_tcp_newreno,
ymax=200
```

C. Host filtering

The analysis functions have a rudimentary filter mechanism. Note, that this mechanism filters only what is plotted, but not what data is extracted. The source_filter parameter allows filtering only flows from specific sources. The filter string format is:

```
(S|D)_<ip>_<port>[;(S|D)_<ip>_<port>]*
```

The following command only plots data for flows *from* host 172.16.10.2 port 80:

```
> fab analyse_all:source_filter=
"S_172.16.10.2_80"
```

Note, that the notion of flow here is unidirectional. Thus in the above example flows from 172.16.10.2 port 80 are shown, but flows to 172.16.10.2 port 80 are not shown. We can also only select flows *to* host 172.16.10.2 port 80 by specifying:

```
> fab analyse_all:source_filter=
"D_172.16.10.2_80"
```

As a side effect, the specified filter string also determines the order of the flows in the graph(s). Flows are plotted in the order of the filters specified. For example, if there are two flows, one from host 172.16.10.2 port 80 and another from host 172.16.10.2 port 81 by default the port 80 flow would be the first data series and the port 81 flow would be the second data series. One can reverse the two flows in the graphs by specifying:

```
> fab analyse_all:source_filter=
"S_172.16.10.2_81;S_172.16.10.2_80"
```

Probably, in the future we should rename this parameter, as it is now a filter on source or destination. Also note that the filter specified is the flows selected (not the flows filtered out).

D. Comparison of metrics depending on variables

The task analyse_cmpexp allows to plot the metrics 'throughput', 'spprtt' and 'tcprtt' (unsmoothed/ERTT) depending on the different experiments for different selected flows. It can show the metric distribution as boxplots (default), or plot the mean or median. Note that analyse_cmpexp relies on data extracted by analyse_all. The following command shows an example, where we plot tcprtt as boxplots:

```
> fab analyse_cmpexp:exp_list=
myexp_list.txt,res_dir="./results/",
variables="run\=0", source_filter=
"D_172.16.10.2_5001;D_172.16.10.3_5006",
metric=tcprtt, lnames="CDG;Newreno"
```

The res_dir parameter is used to specify the results directory (e.g. out_dir used for analyse_all). If it is not specified analyse_cmpexp will execute analyse_all first. The parameter variables can be a semicolon-separated list of variable names (names as used in the log file names) with associate values (separated by an equal sign). This provide a simple filter, as only experiments are considered where the variable(s) had the value(s) specified. Note that the equals (=) must be escape with backslashes, otherwise Fabric will parse these.

The parameter source_filter works as explained above. While not mandatory to specify, in most cases it should be specified to control for which flows the metrics will be plotted. The metric parameter specifies the metric to plot: 'throughput' (default), 'spprtt' or 'tcprtt'. The ptype parameter specifies the plot type: 'box' (default), 'mean', or 'median'. The out_name parameter allows to specify some name that is appended to the end of the file name prefix passed to the plot function. The exp_list parameter allows to specify the list of experiments (as for analyse_all), making it possible to very precisely select which combinations of parameters should be considered. For example, we can remove certain parameter values by removing all test IDs with these values from the list passed via exp_list. The out_dir parameter allows to specify the output directory (same as for the other

functions). The ymax parameter allows to specify a custom maximum value for the y-axis (as for the other plot functions). The lnames parameters allows to specify the legend names used (list of semicolon-separated strings). Note, the number of legend strings must the same as the number of source filters. By default the legend names are the source filters specified.

Currently it is not possible to reorder the different parameters for plotting other than by generating a custom experiment ID list. The default order is the order specified in the config (which is the same as the order in the file names). Also, currently the x-axis labels contains all variable parameters, even the ones that had only one value (and were de-facto constant).

E. Combining graphs

There are two simple shell scripts to combine different result graphs (different PDF files) on a single page for easy comparison. The assumption is that the TCP congestion control algorithm is the innermost parameter varied, or in other words the last varied parameter in the file name.

The script `tcp_comparison.sh` can be used to combine throughput, RTT, CWND or SPP RTT graphs for up to four different TCP congestion control algorithms on one page. For example, the following command creates four PDFs, each with four graphs for each TCP congestion control algorithm (assuming the test ID is 20131220-182929_del_<delay>_tcp_<tcp_algo>):

```
> tcp_comparison.sh 20131220-182929_del_10_tcp
test
```

The output files are:

```
test_cwnd_different_tcps.pdf
test_spprtt_different_tcps.pdf
test_tcprtt_different_tcps.pdf
test_throughput_different_tcps.pdf
```

The second script `tcp_comparison_allinone.sh` creates the same PDFs as above but in addition creates one single-page PDF with all throughput, RTT, CWND and SPP RTT graphs for up to four TCP CC algos (up to 16 graphs in total). It can be used as follows:

```
> tcp_comparison_allinone.sh
20131220-182929_del_10_tcp test
```

The output files are:

```
test_cwnd_different_tcps.pdf
test_spprtt_different_tcps.pdf
test_tcprtt_different_tcps.pdf
test_throughput_different_tcps.pdf
test_different_tcps_allinone.pdf
```

The two scripts are not strictly limited to combining the results for different TCP congestion control algorithms. They can be used with any last parameter as long as there are no more than four values. However, part of the output file names is hard-coded.

To combine graphs with more flexibility one can use the script `combine_graphs.sh`, which is used by the scripts for TCP comparison. The script allows to combine an arbitrary number of graphs one one page. For example, if we want to compare the CWND graphs for two different delay values and four different TCP congestion control algorithms we can do this with the following command:

```
> combine_graphs.sh -c 4x2 -o test.pdf
`find . -name
20131220-182929_del_*_tcp_*cwnd*.pdf | sort`
```

Here the `find` command is used with wildcards to select the PDF files to combine on one page and the `-c` parameter is used to specify that the graphs are organised in a layout with 2 rows and 4 columns. Instead of using `find` one can specify all file names explicitly. This allows for full control of the location of graphs on the single page, but is cumbersome if there are many graphs. Note that `combine_graphs.sh` puts the graphs on the page row after row, i.e. for a 4x2 layout the first four graphs go in the first row, the second four graphs go in the second row, etc.

Note that the scripts for combining graphs require that the pdfjam package is installed (on FreeBSD this can be installed from the ports tree).

VII. Host Control Utility Functions

This section describes a number of utility functions available as Fabric tasks. The fab utility has an option to list all available tasks:

```
> fab -l
```

The exec_cmd task can be used to execute one command on multiple hosts. For example, the following command executes the command `uname -s` on a number of hosts:

```
> fab -H testhost1,testhost2,testhost3
exec_cmd:cmd="uname -s"
```

If no hosts are specified on the command line, the exec_cmd command is executed on all hosts listed in the config file (the union of TPCONF_router and TPCONF_hosts). For example, the following command is executed on all testbed hosts:

```
> fab exec_cmd:cmd="uname -s"
```

The copy_file task can be used to copy a local file to a number of testbed hosts. For example, the following command copies the web10g-logger executable to all testbed hosts except the router (this assumes all the hosts run Linux when the command is executed):

```
> fab -H testhost2,testhost3
copy_file:file_name=/usr/bin/web10g-logger,
remote_path=/usr/bin
```

If no hosts are specified on the command line, the command is executed for all hosts listed in the config file (the union of TPCONF_router and TPCONF_hosts). For example, the following command copies the file to all testbed hosts:

```
> fab copy_file:file_name=
/usr/bin/web10g-logger,remote_path=/usr/bin
```

The authorize_key task can be used to append the current user's public RSA key to the ~./ssh/authorized_keys file of the remote user. The user can then login via SSH without having to enter a password. For example, the following command enables password-less access for the user on all testbed hosts:

```
> fab -H testhost1,testhost2,testhost3
authorize_key
```

Note: the authorize_key task assumes the user has a ~/.ssh/id_rsa.pub key file. This can be created with `ssh-keygen -t rsa`. Also note that the task does not check if the public key is already in the remote user's authorized_keys file, so executing this task multiple times may lead to duplicate entries in the remote user's authorized_keys file.

The init_os task can be used to reboot hosts into specific operating systems (OSs). For example, the following command reboots the hosts testhost1 and testhost2 into the OSs Linux and FreeBSD respectively:

```
> fab -H testhost1,testhost2
init_os:os_list="Linux\,FreeBSD",
force_reboot=1
```

Note that the commas in os_list need to be escaped with backslashes (\) since otherwise Fabric interprets the commas as parameter delimiters. By default force_reboot is 0, which means hosts that are already running the desired OS are not rebooted. Setting force_reboot to 1 enforces a reboot. By default the script waits 100 seconds for a host to reboot. If the host is not responsive after this time, the script will give up unless the do_power_cycle parameter is set to 1. This timeout can be changed with the boot_timeout parameter, which specifies the timeout in seconds (as integer). A minimum boot_timeout of 60 seconds will be enforced.

The do_power_cycle parameter can be set to 1 to force a power cycle if a host does not respond after a reboot. If after the boot_timeout the host is not accessible the script will perform a power cycle. The script will then wait for boot_timeout seconds again for the host to come up. If the host is still not up after the timeout the script will give up (there are no further automatic power cycles). The following command shows an example with do_power_cycle set to 1:

```
> fab -H testhost1,testhost2
init_os:os_list="Linux\,FreeBSD",
force_reboot=1,do_power_cycle=1
```

The power_cycle task can be used to power cycle hosts, i.e. if hosts become unresponsive. After the power cycle the host will boot the last selected OS. For example, the following command power cycles the hosts testhost1 and testhost2:

```
> fab -H testhost1,testhost2 power_cycle
```

The check_host command can be used to check if the required software is installed on the hosts. The task only checks for the presence of necessary tools, but it does not check if the tools actually work. For example, the following command checks all testbed hosts:

```
> fab -H testhost1,testhost2,testhost3
check_host
```

The check_connectivity task can be used to check connectivity between testbed hosts with `ping`. This task only checks the connectivity of the internal testbed network, not the reachability of hosts on their control interface. For example, the following command checks whether each host can reach each other host across the testbed network:

```
> fab -H testhost1,testhost2,testhost3
check_connectivity
```

VIII. Extending the implementation

This section contains some notes on extending the current implementation. We refer to Python functions (which can be Fabric tasks) using the notation of <python_file>.py:<function>().

A. Additional host setup

Any general host setup (e.g. sysctl settings for all experiments) should be added in `hostsetup.py:init_host()`. Note that in this function there are three different sections, one for each OS (FreeBSD, Linux, Windows/Cygwin). Commands that shall only be executed in certain experiments can be set in the config (TPCONF_host_init_custom_cmds).

B. New TCP congestion control algorithm

Adding support for a new TCP congestion control algorithm requires to modify `hostsetup.py:init_cc_algo()`. The new algorithm needs to be added to the list of supported algorithms and in the OS-specific sections code need to be added to load the corresponding kernel module.

C. New traffic generator

Adding a new traffic generator requires to add a new start task in `trafficgens.py`. The current start tasks always consist of two methods, the actual start method is a wrapper around an internal _start method. This is to have the host on which the generator is started as explicit parameter (and not as Fabric hosts parameter) and this also allows to have multiple traffic generators that actually use the same underlying tool (for example this is the case for httperf). The new start method must be added to the imports in `experiment.py`.

The traffic generator start function must, after the traffic generator process has been started, register the started process with its process ID by calling `bgproc.register_proc()`. Then it is ensured that the process will be stopped at the end of the experiment and the traffic generator's log file is collected when `runbg.py:stop_processes()` is called. A current limitation is that there can only be one log file per traffic generator process. Some traffic generators also have stop methods. Initially, the idea was that traffic generators could be started and stopped from the command line directly, but this is not supported at the moment, i.e. some stop methods are not implemented (empty).

D. New data logger

To add a new data logger a start method and possibly a stop method need to be added in `loggers.py`. The new logger's start method should be called from `loggers.py:start_loggers()` via Fabric's execute(), but could also be called from `experiment.py:run_experiment()` if required (in the latter case it must be added to the imports in `experiment.py`).

If the logger is a userspace process, such as tcpdump, at the end of the start function it should register itself (including its process ID) using `bgproc.register_proc_later()`. Then it is ensured that the logging process will be stopped at the end of the experiment and the log file is collected when `runbg.py:stop_processes()` is called. In this case no stop method needs to be implemented.

If the logger is not a userspace process, for example SIFTR on FreeBSD, start *and* stop methods need to be implemented. The start method must still call `bgproc.register_proc_later()`, but the process ID must be set to zero. The stop method must be called from `runbg.py:stop_processes()` if the process ID is zero and the internal TEACUP name of the process is the name of the new logger.

E. New analysis method

To add a new analysis method add an analysis task in `analysis.py`. If the new analysis should be carried out as part of the analyse_all task, the new task must be called from `analysis.py:analyse_all()` via Fabrics `execute()` function. The new task should implement the parameters test_id, out_dir, replot_only, source_filter and min_values (see existing analyse tasks as examples). The new task must be added to the imports in `fabfile.py`.

F. New Linux kernel on hosts

The Linux kernel is hard-coded in `hostsetup.py`. If the hosts are upgraded to a new kernel, which should be booted by default, the names of the old kernel and old inird need to be replaced in `hostsetup.py:init_os()`. In future versions

TEACUP could be extended so that the kernel booted is another parameter in `config.py`.

IX. Known Issues

During the host setup phase the scripts enable and disable NICs. On Windows the enable and disable interface commands have permanent effect. If TEACUP is interrupted between a disable and enable NIC command, and the Windows host is rebooted for the next experiment, the network interface configuration may be lost. In this case the NIC may need to be reconfigured manually.

TEACUP logs all output from the traffic generators, such as iperf or httperf. However, some of these tools only generate output after they completed. If the experiment duration is set shorter than the run-time of the tools, the resulting log file may be empty. Possibly this issue could be mitigated by turning the stdout/stderr buffering off for these tools in future versions.

X. Conclusions and Future Work

In this report we described TEACUP, a Python-based software we developed to run automated TCP performance tests in a controlled testbed. In the future we will continue to extend TEACUP with more features.

Acknowledgements

TEACUP v0.4 was developed as part of a project funded by Cisco Systems and titled "Study in TCP Congestion Control Performance In A Data Centre". This is a collaborative effort between CAIA and Mr Fred Baker of Cisco Systems.

References

[1] A. Finamore, M. Mellia, M. M. Munafò, R. Torres, and S. G. Rao, "Youtube everywhere: Impact of device and infrastructure synergies on user experience," in *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, ser. IMC '11, 2011, pp. 345–360.

[2] A. Rao, A. Legout, Y.-s. Lim, D. Towsley, C. Barakat, and W. Dabbous, "Network characteristics of video streaming traffic," in *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '11, 2011, pp. 25:1–25:12.

[3] "Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats," ISO, 2012, iSO/IEC 23009-1:2012. http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57623.

[4] L. Stewart, "SIFTR – Statistical Information For TCP Research." [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools.html

[5] "The Web10G Project." [Online]. Available: http://web10g.org/

[6] S. Zander, G. Armitage, "CAIA Testbed for TCP Experiments," Centre for Advanced Internet Architectures, Swinburne University of Technology, Tech. Rep. 140314B, 2014. [Online]. Available: http://caia.swin.edu.au/reports/140314B/CAIA-TR-140314B.pdf

[7] "Fabric 1.8 documentation." [Online]. Available: http://docs.fabfile.org/en/1.8/

[8] "Fabric 1.8 documentation – Installation." [Online]. Available: http://docs.fabfile.org/en/1.8/installation.html

[9] "iperf Web Page." [Online]. Available: http://iperf.fr/

[10] "LIGHTTPD Web Server." [Online]. Available: http://www.lighttpd.net/

[11] HP Labs, "httperf homepage." [Online]. Available: http://www.hpl.hp.com/research/linux/httperf/

[12] J. Summers, T. Brecht, D. Eager, B. Wong, "Modified version of httperf," 2012. [Online]. Available: https://cs.uwaterloo.ca/~brecht/papers/nossdav-2012/httperf.tgz

[13] "nttcp-1.47 – An improved version of the popular ttcp program." [Online]. Available: http://hpux.connect.org.uk/hppd/hpux/Networking/Admin/nttcp-1.47/

[14] M. Mathis, J. Heffner, and R. Raghunarayan, "TCP Extended Statistics MIB," RFC 4898 (Proposed Standard), Internet Engineering Task Force, May 2007. [Online]. Available: http://www.ietf.org/rfc/rfc4898.txt

[15] L. Stewart, "SIFTR v1.2.3 README," July 2010. [Online]. Available: http://caia.swin.edu.au/urp/newtcp/tools/siftr-readme-1.2.3.txt

[16] M. Mathis, J. Semke, R. Reddy, J. Heffner, "Documentation of variables for the Web100 TCP Kernel Instrumentation Set (KIS) project." [Online]. Available: http://www.web100.org/download/kernel/tcp-kis.txt

[17] J. Gettys, "Best practices for benchmarking bufferbloat." [Online]. Available: http://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benchmarking_Codel_and_FQ_Codel

[18] Linux Foundation, "netem – Network Emulation Functionality," November 2009. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/netem

[19] ——, "IFB – Intermediate Functional Block device," November 2009. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/ifb

[20] S. Zander and G. Armitage, "Minimally-Intrusive Frequent Round Trip Time Measurements Using Synthetic Packet-Pairs," in *The 38th IEEE Conference on Local Computer Networks (LCN 2013)*, 21-24 October 2013.

[21] A. Heyde, "SPP Implementation," August 2013. [Online]. Available: http://caia.swin.edu.au/tools/spp/downloads.html

[22] D. Hayes, "Timing enhancements to the FreeBSD kernel to support delay and rate based TCP mechanisms," Centre for Advanced Internet Architectures, Swinburne University of Technology, Melbourne, Australia, Tech. Rep. 100219A, 19 February 2010. [Online]. Available: http://caia.swin.edu.au/reports/100219A/CAIA-TR-100219A.pdf

```
import sys
import datetime
from fabric.api import env

env.user = 'root'
env.password = 'password'
env.shell = '/bin/sh -c'
env.timeout = 5
env.pool_size = 10

TPCONF_script_path = '/home/test/src/teacup'
sys.path.append(TPCONF_script_path)
TPCONF_tftpboot_dir = '/tftpboot'
TPCONF_router = [ '192.168.1.4', ]
TPCONF_hosts = [ '192.168.1.2', '192.168.1.3', ]
TPCONF_host_internal_ip = {
'192.168.1.4' : [ '172.16.10.1', '172.16.11.1' ],
'192.168.1.2' : [ '172.16.10.2' ],
'192.168.1.3' : [ '172.16.11.2' ], }

now = datetime.datetime.today()
TPCONF_test_id = now.strftime("%Y%m%d-%H%M%S") + '_experiment'
TPCONF_remote_dir = '/tmp/'
TPCONF_host_os = {
'192.168.1.4' : 'Linux',
'192.168.1.2' : 'FreeBSD',
'192.168.1.3' : 'FreeBSD', }
TPCONF_force_reboot = '1'
TPCONF_boot_timeout = 100
TPCONF_do_power_cycle = '0'
TPCONF_host_power_ctrlport = {}
TPCONF_power_admin_name = ''
TPCONF_power_admin_pw = ''

TPCONF_router_queues = [
( '1', "source='172.16.10.0/24', dest='172.16.11.0/24', delay=V_delay, loss=V_loss, rate=V_urate, queue_disc=V_aqm, queue_size=V_bsize" ),
( '2', "source='172.16.11.0/24', dest='172.16.10.0/24', delay=V_delay, loss=V_loss, rate=V_drate, queue_disc=V_aqm, queue_size=V_bsize" ), ]

traffic_iperf = [
( '0.0', '1', "start_iperf, client='192.168.1.2', server='192.168.1.3', port=5000, duration=V_duration" ),
( '0.0', '2', "start_iperf, client='192.168.1.2', server='192.168.1.3', port=5001, duration=V_duration" ), ]
TPCONF_traffic_gens = traffic_iperf;

TPCONF_duration = 30
TPCONF_runs = 1
TPCONF_ECN = [ '0', '1' ]
TPCONF_TCP_algos = [ 'newreno', 'cubic', 'htcp', ]
TPCONF_host_TCP_algos = { }
TPCONF_host_TCP_algo_params = { }
TPCONF_host_init_custom_cmds = { }
TPCONF_delays = [ 0, 25, 50, 100 ]
TPCONF_loss_rates = [ 0, 0.001, 0.01 ]
TPCONF_bandwidths = [ ( '8mbit', '1mbit' ), ( '20mbit', '1.4mbit' ), ]
TPCONF_aqms = [ 'pfifo', 'codel', 'pie' ]
TPCONF_buffer_sizes = [ 1000, 1 ]

TPCONF_parameter_list = {
'delays' : ( [ 'V_delay' ], [ 'del' ], TPCONF_delays, {} ),
'loss' : ( [ 'V_loss' ], [ 'loss' ], TPCONF_loss_rates, {} ),
'tcpalgos' : ( [ 'V_tcp_cc_algo' ], [ 'tcp' ], TPCONF_TCP_algos, {} ),
'aqms' : ( [ 'V_aqm' ], [ 'aqm' ], TPCONF_aqms, {} ),
'bsizes' : ( [ 'V_bsize' ], [ 'bs' ], TPCONF_buffer_sizes, {} ),
'runs' : ( [ 'V_runs' ], [ 'run' ], range(TPCONF_runs), {} ),
'bandwidths' : ( [ 'V_drate', 'V_urate' ], [ 'down', 'up' ], TPCONF_bandwidths, {} ), }
TPCONF_variable_defaults = {
'V_ecn' : TPCONF_ECN[0],
'V_duration' : TPCONF_duration,
'V_delay' : TPCONF_delays[0],
'V_loss' : TPCONF_loss_rates[0],
'V_tcp_cc_algo' : TPCONF_TCP_algos[0],
'V_drate' : TPCONF_bandwidths[0][0],
'V_urate' : TPCONF_bandwidths[0][1],
'V_aqm' : TPCONF_aqms[0],
'V_bsize' : TPCONF_buffer_sizes[0], }


TPCONF_variable_defaults TPCONF_vary_parameters = [ 'tcpalgos', 'delays', 'loss', 'bandwidths', 'aqms', 'bsizes', 'runs', ]
```

Figure 14. Example config.py file