TEACUP v1.0 – A System for Automated TCP Testbed Experiments

Sebastian Zander, Grenville Armitage
Centre for Advanced Internet Architectures, Technical Report 150529A
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 video streaming from YouTube and Netflix is TCP-based [1], [2] and there is an ISO standard for 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 shedding more light on this issue. This report describes TEACUP (TCP Experiment Automation Controlled Using Python) version 1.0 – 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 delays and/or loss rates, 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 and TCP stack information.**

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

CONTENTS

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 video streaming from YouTube and Netflix is TCP-based [1], [2] and there is an ISO standard for 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 shedding more light on this issue.

This report describes TEACUP (TCP Experiment Automation Controlled Using Python) version 1.0 – a software tool for running automated TCP experiments in a testbed. It updates the previous report describing TEACUP version 0.9 [4]. The TEACUP project originated at Swinburne University of Technology's Centre for Advanced Internet Architectures (http://caia.swin.edu.au/tools/teacup), and from version 1.0 the source code is freely available on SourceForge at http://sourceforge.net/projects/teacup

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 delays and/or loss rates, 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 [5] and Web10G [6] logs. A related technical report [7] describes the design and implementation of a specific testbed designed to be controlled by TEACUP.

TEACUP also provides a number of tasks for the analysis of data collected during experiments. The analysis aspect of TEACUP is quite distinct from the tasks that actually run experiments and gather testbed data, hence these are described in the accompanying technical report [8].

This report is organised as follows. Section II describes the overall design of TEACUP – the testbed network model, process flow and use of Fabric. Section III outlines the traffic generators and logging available under TEACUP, while host and bottleneck router configuration is summarised in Section IV. Section V describes the configuration options available when running experiments as described in Section VI. Section VII describes utility functions that can be used for host maintenance. Section IX outlines how to extend TEACUP. Section X lists known issues. Section XI concludes and outlines future work.

II. TEACUP Requirements and 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. Finally, we describe the naming scheme for log files.

A. Requirements

The following paragraphs describe the requirements that motivated our design of TEACUP.

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

2) Topology: TEACUP runs experiments in a controlled environment like that shown in Figure 1 [7], where one bottleneck router interconnects two *experiment networks*. The experiment networks contain hosts that can act as traffic sources and sinks. The router, all hosts and TEACUP (on a control server) are also connected to a separate *control network*.[1] TEACUP configures hosts before each experiment and collects data from hosts after each experiment.[2] Where the control network is a private network, the control server may act as a NAT gateway enabling all testbed hosts to access the public Internet if necessary.

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.

[1]The control server also runs a DHCP+TFTP server for the PXE boot setup described in [7].

[2]If the networks are implemented as VLANs on a suitable Ethernet switch, TEACUP can also automate the assignment of hosts to one or the other experiment network and associated VLAN (Section V-H).

4) Path characteristics: The system must be able to 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. Optionally other AQM mechanisms may be supported. Since FreeBSD does not support some of the required AQMs the router must run Linux (but to allow comparison TEACUP also has some basic support for a FreeBSD router). The buffer size must be configurable.

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

7) Host OS: We are OS-agnostic. However, to cover the various TCP algorithms *and* their common implementations TEACUP must support scenarios where sources and/or destinations are Windows (Compound), Linux (NewReno, CUBIC), FreeBSD (NewReno, CUBIC, CDG, HD) or Mac OS X (NewReno). Cygwin is used to instrument Windows [7].

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

TEACUP is built on Fabric [9] (Section II-D), a Python (2.5 or higher) library and command-line tool for streamlining the remote application deployment or system administration tasks using SSH. Our design is based on multiple small tasks that are combined to run an experiment or a series of experiments (where some may also be executed directly from the command line). Functions which are not Fabric tasks are ordinary Python functions. Currently, we do not make use of the object-oriented capabilities of Python.

Figure 2 shows the main functional building blocks. All the blocks in the diagram have corresponding Python files. However, we have summarised a number of Python

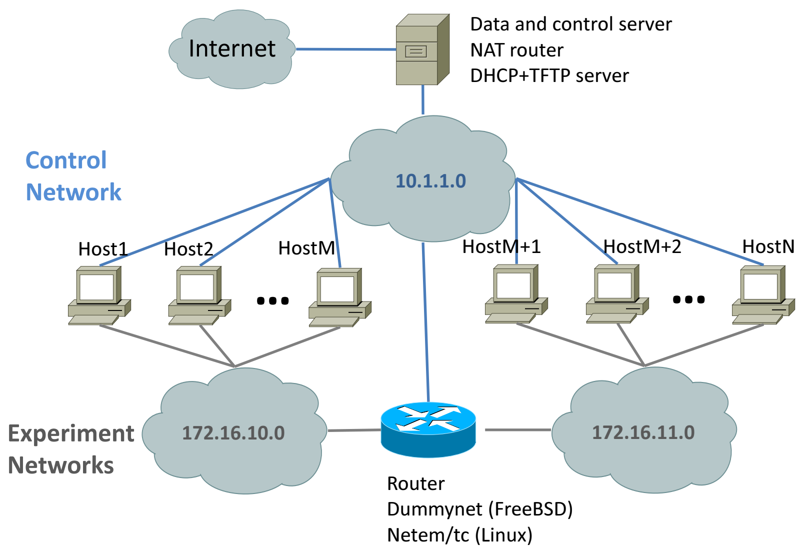


Figure 1: Testbed topology

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 a series of similar experiments with different parameters. The fabfile block also provides access to all other tasks via Python 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 that are still running. The host_setup block implements all functions to setup networking on the testbed hosts (including basic setup of the testbed router). The router_setup block implements the functions that set up 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) Topology reconfiguration (assign hosts to subnets)
   5) Get host information again: OS, NIC names, NIC MAC addresses
   6) Run sanity checks
      - Check that tools to be used exist
      - Check connectivity between hosts
      - Kill any leftover processes on hosts
   7) Initialise hosts
      - Configure NICs (e.g. disable TSO)
      - Configure ECN use
      - Configure TCP congestion control
      - Initialise router queues
   8) Configure router queues: set router queues based on config

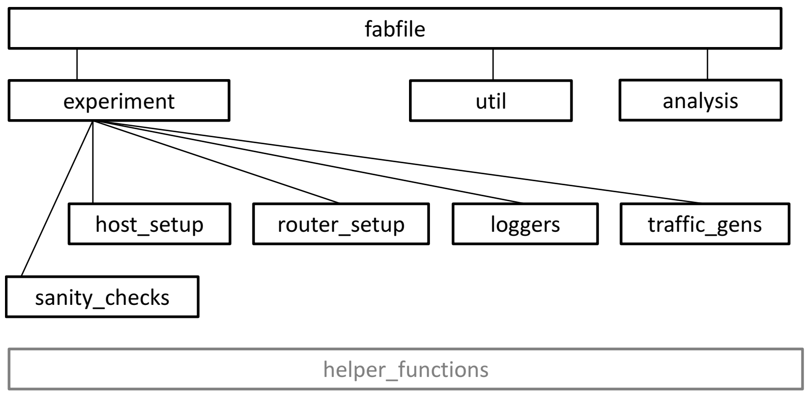


Figure 2: TEACUP main functional blocks

9) Log host state: log host information (see Section III-C)
10) Start all logging processes: tcpdump, SIFTR/Web10G etc.
11) Start all traffic generators: start traffic generators based on config
12) Wait for experiment to finish
13) Stop all running processes on hosts
14) Collect all log files from logging and traffic generating processes
15) Log experiment test ID in file `experiments_completed.txt`

IV) If we have another parameter combination to run go to step III, otherwise finish

D. Fabric – overview and installation

Fabric [9] 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`, which is 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. A Fabric task may also execute another Fabric task with Fabric's `execute()` function.

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

TEACUP was developed with Fabric versions 1.8–1.10, but it should also 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 [10].

III. Traffic Generation and Logging

This section describes the implemented traffic sources/sinks and loggers, the range of information logged and the log file naming schemes.

A. *Traffic sources and sinks*

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

1) iperf: The tool iperf [11] 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 [12] web server is started. This can be used as traffic source for httperf-based sinks. There are also scripts 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.

4) httperf: The tool httperf [13] 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 [13]).

5) httperf_dash: This starts a TCP video streaming httperf client [14] that emulates the behaviour of DASH 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 3 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. Note that emulating the incast problem in a physical testbed is difficult, if the number of hosts (number of responders) is relatively small.

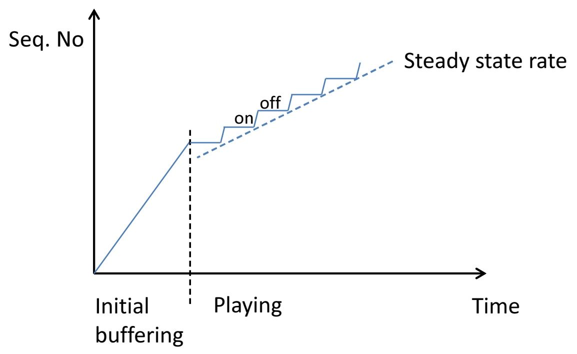


Figure 3: TCP video streaming (DASH) behaviour

7) nttcp: This starts an nttcp [15] client and an nttcp server for 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 only a few packets are exchanged before and after the data flow.

B. *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 [5]. On Linux we use Web10G [6], which implements the TCP EStats MIB [16] inside the Linux kernel, with our own logging tool based on the Web10G library. For Windows 7 we implemented our own logging tool, which can access the TCP EStats MIB inside the Windows 7 kernel. SIFTR has not been ported to Mac OS X, so for Mac OS X we implemented our own logging tool that outputs TCP statistics logs in SIFTR format based on DTrace [17]. Note that the DTrace tool collects most but not all statistics collected by SIFTR (the tool's documentation describes the limitations).

The statistics collected by SIFTR are described in the SIFTR README [18]. The statistics collected by our Web10G client and the Windows 7 EStats logger are described as part of the web100 (the predecessor of Web10G) documentation [19].

C. 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 (where <test_id_prefix> is a test ID prefix and <test_id> is a test ID):

- <test_id>_ifconfig.log.gz: This file contains the output of `ifconfig` (FreeBSD, Linux or MacOSX) 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, Linux, MacOSX or Windows with NTP daemon installed) or `w32tm` (Windows).
- <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, Linux or MacOSX) 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 V). 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 about the TCP congestion control algorithm used on each host, and any TCP parameters specified.
- <test_id>_tcpmod.log.gz: This file contains the TCP congestion control kernel module parameter settings (Linux only).
- <test_id>_ethtool.log.gz: This file contains the network interface configuration information provided by `ethtool` (Linux only).
- <test_id_prefix>_nameip_map.log.gz: This file logs host names and IP addresses (control interface IP addresses) of all hosts and routers participating in the series of experiments.

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) as well as 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.

D. Config information logged

TEACUP also logs information about the configuration of each series of experiments and variables set for each experiment. The following information is gathered *before* an experiment is started (where <test_id_prefix> is a test ID prefix and <test_id> is a test ID):

- <test_id>_config_vars.log.gz: This file logs the values of all V_ variables for each experiment (see Section VI-D).
- <test_id_prefix>_config.tar.gz: This file contains copies of the config file(s) – there can be multiple files if Python's execfile() is used to separate the configuration into multiple files.
- <test_id_prefix>_tpconf_vars.log.gz: This file lists all TPCONF_ parameters specified in the config file(s) in a format that can be imported into Python.
- <test_id_pfx>_varying_params.log.gz: This file contains a list of all V_ variables varied during the experiment(s) and their short names used in file names (see Section VI-D).

E. Log file naming

The log 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 VI).

The `[<par_name>_<par_val>_]*` is the zero to n parameter names and parameter values (separated by an underscore). Parameter names (`<par_name>`) should not contain underscores by definition and all underscores in parameter values (`<par_val>`) are changed to hyphens (this allows later parsing of the names and values using the underscores as separators). 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 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 special 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 the 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 compressed with gzip, hence their file names end with '.gz'.

Figure 4 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.

All log files for one experiment (e.g. fab run_experiment_single) or a series of experiments (e.g. fab run_experiments_multiple) are stored under a sub directory named `<test_ID_pfx>` created inside the directory where fabfile.py is located.[3]

[3]Prior to version 0.4.7 TEACUP stored all log files in the directory where fabfile.py was located.

IV. Host and Router Configuration

This section provides a general description of how the hosts and router are configured for each experiment and test within an experiment. The router implements a bottleneck with configurable one-way delays, rate limits and AQM (active queue management).

A. 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 or disabled 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).

B. 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. In the following two sub sections we first describe our overall approach to setup rate limiting, AQM and delay/loss emulation for the Linux router. Then, 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). To filter packets and direct them into the 'pipes' we use netfilter [20]. The hierarchical token bucket (HTB) queuing discipline is used for rate limiting with the desired AQM queuing discipline (e.g. pfifo,

```
# 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 4: Example file names

codel) as leaf node (this is similar to a setup mentioned at [21]). After rate shaping and AQM, constant loss and delay is emulated with netem [22]. For each pipe we set up 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 10):

1) Arriving packets are marked at the netfilter mangle table's POSTROUTING hook depending on source and destination IP address with a unique mark for each pipe.[4]
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 [23].
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 the 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 plus pfifo are used as leaf queuing discipline.[5]
6) Packets leave the router via the stack and network card driver.

[4]There also is a dummy rule "MARK and 0x0" inserted first, which is used 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.

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

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 the same interface, but then there are certain limitation, such as netem must be before the AQM and [21] reported that in such a setup netem causes problems. Also, a big advantage with our setup is that it is possible to emulate different delay or 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 5 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 6, 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 7 (again class 1:1 is set if the mark is 0x1).

On ifb1 we use the queuing discipline setup as shown in Figure 8. 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 9.

After leaving netem the packets are passed to the stack which then passes them to the NIC driver. For the sake of brevity we are not describing the downstream direction here, but the principle is exactly the same. The only

```
> 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 5: 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 6: 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 7: 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 8: Queuing discipline setup on pseudo interface

differences are the interfaces used (eth2 and ifb0 instead of eth3 and ifb1) and the different HTB, AQM and netem parameters.

Figure 10 shows the flow of packets with the different steps carried out in the order of the numbers in parenthesis. The marking/classifying is not shown explicitly, 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 or loss for different flows that share the same bottleneck/AQM. Multiple tc filters on the ifb interface can classify different flows as the same class so they share the same bottleneck. However, on the eth interface we can 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 [21] the HTB queuing discipline has a built-in buffer of one packet (that cannot be changed) and the device drivers also have separate buffers.

C. *FreeBSD router setup*

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

```
> 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 9: Queuing discipline setup on outgoing interface (netem)

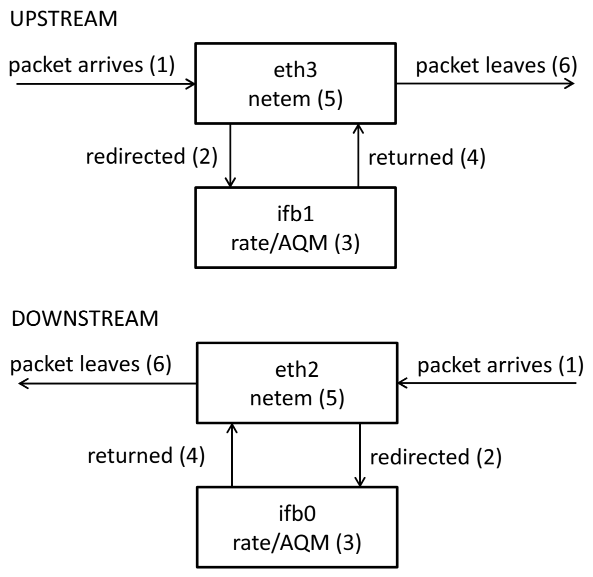


Figure 10: Flow of packets through our queue setup

On FreeBSD each pipe is realised as one Dummynet pipe, which does the rate shaping, loss/delay emulation 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 <num> ip from <source> to <dest> out" rule. If a pipe is bidirectional there is an additional "pipe <num> ip from <dest> to <source> out" rule. The pipe number <num> is automatically determined by TEACUP. A more sophisticated setup for FreeBSD remains future work.

V. Config File

This section describes TEACUP's top-level config.py file that controls the experiments.

A. *Config file location*

By default TEACUP will load the config.py file that is in the directory where fabfile.py is located – the directory from which experiments are started. Since TEACUP version 0.9, alternative config files can be specified using fab's `--set` parameter. For example, if we want to run a series of experiments with the configuration in myconfig.py, we simple need to run TEACUP as follows:

```
> fab --set teacup_config=myconfig.py
run_experiment_multiple
```

B. *V_variables*

To iterate over parameter settings for each experiment TEACUP uses *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 V-L), traffic generator settings (see Section V-N), TCP algorithm settings (see Section V-O) or host setup commands (see Section V-K). Section V-Q describes how to define V_variables.

C. *Fabric configuration*

The following settings in the config file are Fabric settings. For a more in-depth description refer to the Fabric documentation [9]. All Fabric settings are part of the Fabric `env` dictionary and hence are 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 (and the corresponding private key on the control host is ~/.ssh/id_rsa or a file <key_file> specified with fab -i <key_file> or the env.key_filename configuration parameter [9]).

```
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
```

D. Testbed configuration

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

TPCONF_script_path specifies the path to the TEACUP scripts, and is appended to the Python path.

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

Two lists specify the testbed hosts. TPCONF_router specifies the list of routers. Note that prior to TEACUP version 0.9 the TPCONF_router list was limited to only *one* router. Since version 0.9 a list of routers can be specified. TPCONF_hosts specifies the list of hosts. Router and 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'],
}
```

E. Sanity checks settings

By default TEACUP checks the connectivity between testbed hosts by performing a ping from each host to each other host that should be reachable (according to the router configuration and IP address configuration of the hosts). This check is carried out before each experiment. During a series of experiments the testbed is likely in a stable condition and the repeated connectivity check may be unnecessary. It can be turned off by setting the variable TPCONF_check_connectivity to '0', i.e. adding the following line to the config file:

```
TPCONF_check_connectivity = '0'
```

F. General experiment settings

TPCONF_test_id specifies 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/'
```

TEACUP performs a very simple time synchronisation check at the start (after waiting for more than 30 seconds after the hosts were rebooted). It checks the time offset between the control host (the host that executes TEACUP) and each testbed host listed in the config file. TPCONF_max_time_diff specifies the maximum tolerable clock offset, i.e. the maximum allowed time difference in seconds. If the clock offset for one of the testbed hosts is too large, TEACUP will abort the experiment or series of experiments.

```
TPCONF_max_time_diff = 1
```

TPCONF_debug_level specifies the debug level for experiments. At the default level of 0, no debug information is generated. If the variable is set to a higher value debug information is generated, for example at a debug level of 1 .Rout files are generated when plotting graphs.

```
TPCONF_debug_level = 0
```

The parameter TPCONF_web10g_poll_interval allows to specify the poll interval in milliseconds for web10g loggers on Linux or Windows. The minimum value is 1 ms and the maximum value is 1000 ms. The default value is 10 ms. Note that the control of the interval length is not very accurate and with small intervals less then 10 ms, the actual interval will likely be larger than the specified interval.

```
TPCONF_web10g_poll_interval = 10
```

G. tcpdump/pcap configuration

The variable TPCONF_pcap_snaplen sets the *snap length* (number of bytes captured by tcpdump per Ethernet frame). If not specified the default is 80 bytes. Setting TPCONF_pcap_snaplen=0 means 'capture all bytes'. The following shows an example where we set the snap length to 128 bytes.

```
TPCONF_pcap_snaplen = 128
```

H. Topology configuration

Since version 0.8 TEACUP can automate the assignment of each host into one of the two subnets. The topology configuration is *EXPERIMENTAL* and based on a number of assumptions around the network setup.[6] For an experiment or a series of experiment the topology (re)configuration is (optionally) carried out after the machines haven been rebooted and before any sanity checks are done.

Automatic topology configuration is enabled by setting the variable TPCONF_config_topology to '1'. (If this variable is undefined or set to '0' there is no topology configuration.)

```
TPCONF_config_topology = '1'
```

[6]The most critical restriction is that switch reconfiguration has only been tested with Dell 5324 and Dell N3000 series switches.

When topology configuration is enabled TEACUP actively *(re)configures* each host's test subnet IP address to that specified in TPCONF_host_internal_ip, then *changes* the VLAN configuration of the testbed switch's ports so all hosts are connected to the right subnets.

Automated VLAN (re)configuration requires SSH access to the switch, with the same SSH user name and password as used on all other hosts. As many switches limit the number of concurrent SSH sessions allowed, TEACUP currently performs the configuration of the ports on switch sequentially (host by host). However, after the switch has been configured TEACUP carries out the setup of each host (network interface and routing configuration) in parallel to reduce the overall time for topology configuration.

When mapping switch ports to VLANs, TEACUP assumes we are using /24 subnets and that the third octet of the IP address is identical to the VLAN name configured on the switch. For example, a host being configured with address 172.16.10.2 is mapped to a corresponding VLAN named '10' on the switch..

TEACUP also assumes that all hosts are differentiated by appending consecutive numbers to their hostnames. The lowest number, h, can be chosen arbitrarily. For example, if we have two hosts they could be named 'testhost1' and 'testhost2'. TEACUP further assumes that hosts are connected to consecutive ports of the switch in the same order as their hostname numbering. For example, if we have testhost1 and testhost2, and testhost1 is connected to switch port n then testhost2 must be connected to switch port $n + 1$.

The address or host name of the switch can be configured with the variable TPCONF_topology_switch. To map a host to the corresponding port the variables TPCONF_topology_switch_port_prefix and TPCONF_topology_switch_port_offset (also called o) can be defined. The name of the switch port used will be the string specified in TPCONF_topology_switch_port_prefix concatenated with the the number $h + o$. The following shows the default configuration.

```
TPCONF_topology_switch = 'switch2'
TPCONF_topology_switch_port_prefix = 'Gi1/0/'
TPCONF_topology_switch_port_offset = 5
```

The above configuration assumes that testhost1 is connected to switch port 'Gi1/0/6' and testhost2 is connected to switch port ‘Gi1/0/7’ on a switch with the host name ‘switch2’ (which has SSH configuration enabled).

Topology configuration works with all supported OS (Linux, FreeBSD, Windows/Cygwin and Mac OS X). However, the switch reconfiguration code has only been tested with Dell 5324 and Dell N3000 series switches.

Note that the network interfaces on the hosts are currently hard-coded inside TEACUP. For example, for FreeBSD hosts the topology setup method assumes that em1 is the testbed interface. The interface names can only be changed by modifying the Python code.

1) Link speed selection: As of version 0.9, TEACUP can set the link speed of the Ethernet links between the switch and testbed NICs. There are four settings for the speed: ‘10’ (10 Mbps, 10baseT), ‘100’ (100 Mpbs, 100baseT), ‘1000’ (1000 Mbps, 1000baseT), and ‘auto’ (default, which should result in 1000baseT). The variable TPCONF_linkspeed allows to configure the link speed for all hosts. For example, if all hosts should use a link speed of 100 Mbps (100baseT) we can specify

```
TPCONF_linkspeed = '100'
```

However, we can also configure host-specific link speeds with the dictionary TPCONF_host_linkspeed. Each entry must have as index a host name (as defined in TPCONF_router or TPCONF_hosts) and as value one of the possible speed settings explained above. For example, if testhost1 should be set 100 Mbps and testhost2 to 1000 Mbps we can define

```
TPCONF_host_linkspeed = {
        'testhost1' : '100',
        'testhost2' : '1000'
}
```

TPCONF_host_linkspeed entries always overrule TPCONF_linkspeed. If neither variable is specified the default link speed setting is ‘auto’, which should result in 1000baseT assuming Gigabit Ethernet NICs.

Note that the link speed setting is persistent for Linux, FreeBSD and Windows. However, for Mac OS X the speed will reset to 1000baseT after a reboot (normally this should not be an issue as hosts are only rebooted before experiments).

I. Rebooting, OS selection and power cycling

With suitable external support, TEACUP enables automated rebooting of testbed hosts into different operating systems and forced power cycling of hosts that have become unresponsive.

1) PXE-based rebooting: TEACUP’s PXE-based rebooting process is explained in [7].

The IP address and port of the TFTP or HTTP server that serves the .ipxe configuration files during the PXE boot process is specified with the parameter TPCONF_tftpserver. Both IP address and port number must be specified separated by a colon. For example:

```
TPCONF_tftpserver = '10.1.1.11:8080'
```

The path to the directory that is served by the TFTP or HTTP server is specified with TPCONF_tftpboot_dir.

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

Omitting TPCONF_tftpboot_dir, or setting it to an empty string, disables PXE booting. TPCONF_host_os and TPCONF_force_reboot (see below) are then ignored.

2) OS and kernel selection: 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 (any host specified in TPCONF_host_os must be specified in either TPCONF_router or TPCONF_hosts). The current code does simple string matching, it does *not* attempt to resolve host identifiers in some canonical form.

TEACUP currently supports selecting from four different types of OS: ‘Linux’, ‘FreeBSD’, ‘CYGWIN’ (Windows) and ‘Darwin’ (Mac OS X). Selecting the specific Linux kernels to boot is supported with the TPCONF_linux_kern_router and TPCONF_linux_kern_hosts parameters (see below). The OS selection occurs once during reboot, and cannot subsequently be varied during a given experiment.

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

TPCONF_linux_kern_router specifies the Linux kernel booted on the router, and TPCONF_linux_kern_hosts specifies the Linux kernel booted on all other hosts. The name is the kernel image name minus the starting ‘vmlinuz-’, so the name starts with the version number. It is also possible to specify the keywords ‘running’ or ‘current’ to tell TEACUP to use the same kernel that is currently running on the host (this *requires* that the host is already running Linux).

```
TPCONF_linux_kern_router = '3.14.18-10000hz'
TPCONF_linux_kern_hosts = '3.9.8-web10g'
```

The parameter TPCONF_os_partition allows us to specify the partitions for the different operating systems on the hosts (in GRUB4DOS format since our TEACUP-based testbed uses GRUB4DOS to reboot into the desired OS). For example, if we have the configuration below then TEACUP will attempt to boot from the first partition of the first disk if Windows/Cygwin is selected, boot from the second partition of the first disk if Linux is selected, and boot from the third partition of the first disk if FreeBSD is selected.

```
TPCONF_os_partition = {
'CYGWIN'  : '(hd0,0)',
'Linux'   : '(hd0,1)',
'FreeBSD' : '(hd0,2)',
}
```

3) Boot behaviour and timeout: 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 (or kernel in case of Linux) is *not* the desired OS (or kernel in case of Linux), as specified in TPCONF_host_os (and TPCONF_linux_kern_router or TPCONF_linux_kern_hosts), 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
```

4) Forced power cycling: If TEACUP-supported power controllers are installed and TPCONF_do_power_cycle is set to ‘1’, a host is power cycled if it does not reboot within TPCONF_boot_timeout seconds. If TPCONF_do_power_cycle is omitted or set to ‘0’ there is no power cycling.

```
TPCONF_do_power_cycle = '0'
```

If TPCONF_do_power_cycle=1 then parameters TPCONF_power_ctrl_type, TPCONF_host_power_ctrlport, TPCONF_power_admin_name and TPCONF_power_admin_pw must also be set.

TPCONF_power_ctrl_type must be used to specify the type of power controller – ‘9258HP’ (for an “IP Power 9258HP”) or ‘SLP-SPP1008’ (for a “Serverlink SLP-SPP1008-H”). The default is ’9258HP’. A mix of different types of power controllers is currently not supported.

```
TPCONF_power_ctrl_type = '9258HP'
```

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 controller’s 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
```

J. Clock offset measurement (broadcast pings)

All the participating machines should have synchronised clocks (for example by running NTP) so we can accurately compare data measured on different hosts. However, clocks on consumer-grade PC hardware drift even while using NTP. TEACUP provides an additional mechanism to evaluate the quality of the time synchronisation and (optionally) correct for clock offsets in the post-analysis.

When TPCONF_bc_ping_enable is set to ‘1’, TEACUP configures the router host to send a broadcast or multicast ping over the control network. These ping packets will be multicasted or broadcasted to the control interfaces of all other hosts (almost) simultaneously.

During experiments there is no other traffic on the control network and typically network jitter introduced by the sender (router), the network switch or the receiver is small. Thus one can estimate the offsets of the clocks of different hosts with relatively high accuracy by comparing the arrival times of the broadcast or multicast ping packets. Here we explain how to enable and configure the broadcast pings.

TPCONF_bc_ping_enable must be set to '1' to enable the broadcast/multicast ping (default is '0'). TPCONF_bc_ping_rate controls the rate in ping packets per second (default is one packet per second). TPCONF_bc_ping_address is the address the ping is sent to. This must be either a multicast address (e.g. 224.0.1.199) or a broadcast address (e.g. 192.168.0.255 if the control interfaces are in the 192.168.0.0/24 subnet). The following shows an example configuration for enabled multicast pings.

```
TPCONF_bc_ping_enable = '1'
TPCONF_bc_ping_rate = 1
TPCONF_bc_ping_address = '224.0.1.199'
```

In technical report [8] we explain TEACUP's functions to analyse the clock offsets and use them for the correction of timestamps.

K. 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.

The custom commands are executed before the router configuration. So when using for example a FreeBSD router we can use TPCONF_host_init_custom_cmds to increase the maximum allowable Dummynet queue length (using sysctl) before Dummynet is actually configured.

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 an example where we simply execute the command 'echo TEST' on host testhost1.

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

L. Router queue setup

The variable TPCONF_router_queues specifies the router pipes (also referred to as queues here). 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 defined in packets for most queuing disciplines, but for some queuing disciplines it needs to be specified 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 size is specified 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 destination, whereas a bidirectional queue will get the traffic from source to dest *and* from destination 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 or 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 twice the 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 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 11 shows an example queue setup with the same delay and loss for every host and the same delay and loss in both directions (all the parameters are variables here).

Figure 12 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 172.16.11.2, but in both directions it experiences twice the delay.

Since version 0.9 TEACUP supports multiple routers. If multiple routers are specified in TPCONF_router, but there is only one TPCONF_router_queues specification (for example the one shown above), TEACUP will apply the single TPCONF_router_queues specification to all routers. However, in many cases with multiple routers we want to specify router-specific queue setups. This can be done by making TPCONF_router_queues a dictionary with the keys being router names (as specified in TPCONF_router) and the values being TPCONF_router_queues specifications.

As an example let us assume there are two routers: testrouter1 and testrouter2. For each of the routers we need to specify the queue setup (for brevity we use a

```
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 11: 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 12: Router queue definition with attach_to_queue example

placeholder here instead of showing the actual queue specifications:

```
testrouter1_queues = [<queue_spec1>]
testrouter2_queues = [<queue_spec2>]
```

Then we need to define TPCONF_router_queues as explained above:

```
TPCONF_router_queues = {}
TPCONF_router_queues['testrouter1'] =
         testrouter1_queues
TPCONF_router_queues['testrouter2'] =
         testrouter2_queues
```

M. 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 time of the experiment. The second value of the tuple is a unique ID. The third value of the tuple is a list of strings with 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 V-N.

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 TEACUP 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 mathematical operations, such as addition or multiplication, with constants. For example, if a variable 'V_delay' exists one can specify '2·V_delay' as parameter value.

Figure 13 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 files in the source code distribution contain more examples of setting up traffic generators.

N. Available traffic generators

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

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

```
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 13: Traffic generator example

- **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)
- **kill**: By default this is '0' and the iperf client will terminate after duration seconds. If this is set to '1', the iperf client will be killed approximately 1 second after duration. This is a work-around for a "feature" in iperf that prevents it from stopping after the specified duration. (If set to '1', the iperf server is also killed approximately 2 seconds after duration.)
- **mss**: TCP maximum segment size (passed to iperf -M option)
- **buf_size**: Send and receive buffer size in bytes (passed to iperf -j and -k, which only exist for iperf with the CAIA patch [7])
- **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 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, MacOSX default: /opt/local/www/htdocs, 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, MacOSX default: /opt/local/www/htdocs, 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 lengths 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, MacOSX default: /opt/local/www/htdocs, 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 (passed 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)
- **with_timeout**: if set to '0' there is no timeout for requests (a request may take longer to complete than cycle); if set to '1' httperf will close the connection and terminate the session if the download of a chunk takes longer than cycle
- **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.

If prefetch_timeout is set, the prefetch must be completed within the timeout, or otherwise httperf will close the connection and terminate the session. Similarly, if with_timeout='1', httperf will close the connection and terminate the session if the download of a chunk takes longer than cycle seconds.

Note that the number of chunks requested by the httperf client is set to floor((duration - prefetch_timeout) / cycle). This number guarantees that httperf can complete and generate the statistics output within the specified duration, even if the downloading takes the maximum amount of time (downloading always just finishes within the timeouts). However, this also means that if the prefetch is much faster than prefetch_timeout and/or the download of the last chunk is much faster than cycle seconds, httperf will finish well before duration seconds.

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

- **client**: IP or name of client
- **servers**: comma-separated list where each entry specifies one server in the form of '[IP|name]:port'. IP or name is a server's IP address or name and port is the port the server is listening on (a separate session for each server is created via httperf's session log).
- **duration**: duration of incast session in seconds
- **period**: period between requests in seconds (floating point number)
- **burst_size**: number of queries sent at each period start (this is to increase the number of queries in

a testbed that only has a few physical responder machines)
- **response_size**: size of response from each responder in kB
- **extra_params**: Command line parameters passed to httperf

Note that if httperf can not complete a request within period time, if will close the connection and terminate the session with the responder (sessions with other responders are unaffected).

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 the interval TEACUP computes the number of buffers 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 only a few packets are exchanged before and after the data flow.

10) start_httperf_incast_n: This starts an httperf client (querier) and n HTTP servers (responders) for an incast scenario. It also generates the fake content for all n servers. Basically this generator is a shortcut to setting up n servers with n start_http_server and create_http_incast_content. It has the following parameters:

- **client**: IP or name of client
- **servers**: comma-separated list of servers, where each entry is either the IP or the host name of an HTTP server (a separate session for each server is created via httperf's session log)
- **server_port_start**: the port the first server is listening on. All further servers will be started on consecutive port numbers in the order they are specified in servers
- **duration**: duration of incast session in seconds
- **period**: period between requests in seconds (floating point number)
- **burst_size**: number of queries sent at each period start (this is to increase the number of queries in a testbed that only has a few physical responder machines)
- **response_size**: size of response from each responder in kB
- **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, MacOSX default: /opt/local/www/htdocs, Linux/CYGWIN default: /srv/www/htdocs)
- **sizes**: comma-separated list of content file sizes on the servers
- **num_responders**: The querier will only send queries to the first num_responders servers. This can be used to vary the number of responders in a series of experiments
- **extra_params**: Command line parameters passed to httperf

Note that for this method the unique ID is the first ID used. The total number of IDs used is one plus the number of server/responders. Each server will be assigned a consecutive ID number starting with the first ID. The client will have an ID one higher than the highest server ID. If further traffic generators are specified after this generator, the user has to make sure there is no ID number clash.

11) start_fps_game: This starts an emulated FPS game traffic session. From a logical viewpoint this starts several game clients and one game server. However, since the pktgen tool from [24] we use to generate the game traffic can only generate a single unidirectional traffic flow, this task starts pktgen twice as many times as there are clients (half of these processes generate the client-to-server traffic and half of these processes generate the server-to-client traffic). The task has the following parameters:

- **clients**: comma-separated list of clients where each client is specified as name/IP

followed by a colon and the port number (e.g. client1:10000,client2:10001)
- **server**: server name/IP followed by a colon and the server port number (e.g. server1:27960)
- **game_type**: the type of game (can be 'q3', 'hl2cs', 'hl2dm', 'hlcs', 'hldm', 'et2pro', 'q4')
- **c2s_interval**: time interval between packets send by client (in seconds)
- **c2s_psize**: base packet size of packets send by clients (number of bytes of UDP payload); note that pktgen randomly varies the size of packets within a few bytes above the base size)
- **s2c_interval**: time interval between packets send by server (in seconds)
- **duration**: duration of the game session (in seconds)
- **client_start_delay**: delay between the last server-to-client pktgen process started and the first client-to-server pktgen process started (in seconds)
- **extra_params_client**: extra parameters passed to client pktgen processes
- **extra_params_server**: extra parameters passed to server pktgen processes

Note that a pktgen server process will wait for packets from its corresponding client process before starting to send packets. In contrast, a client process will send packets to a corresponding server process immediately. If a client process is started, but the corresponding server process is not listening for packets yet, the server machine will return ICMP port unreachable messages to the client machine, which in turn will terminate the client process. The parameter client_start_delay must be set large enough so that all server processes are started and listen for packets before any client processes are started. The delay must be chosen based on the number of game clients (as each game client results in one pktgen process started on the server). By default client_start_delay is set to three seconds, which is sufficient for up 8 game clients.

Note that pktgen does only support games with 4–32 clients. The smallest allowed number of clients is 4 and the largest allowed number of clients is 32.

Also note that for this method the unique ID is the first ID used. In total the number of IDs used is twice the number of clients. Each server-to-client pktgen will be assigned a consecutive ID number starting with the first ID. Each client-to-server pktgen will be assigned a consecutive ID number starting with an ID one higher than the highest server-to-client pktgen ID. If further traffic generators are specified after this generator, the user has to make sure there is no ID number clash.

O. Mandatory experiment variables

We now describe the mandatory experiment variables that must be in every config file. There are two types: 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).

1) Traffic duration: The duration of the traffic 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) plus some warmup time. TPCONF_duration is specified as follows:

```
TPCONF_duration = 30
```

2) Number of repeats (runs): The number of repetitions (runs) carried out for each unique parameter combination is specified with TPCONF_runs:

```
TPCONF_runs = 1
```

3) Enabling ECN on hosts: TPCONF_ECN specifies whether ECN is used on the hosts that are the traffic sources/sinks. 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 only possible with custom commands.

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

(Note that TPCONF_ECN only enables ECN on the hosts. For full ECN support, the AQM mechanism on the router must also be configured to use ECN.)

4) Congestion control algorithms: TPCONF_TCP_algos specifies the TCP congestion algorithms used. The following algorithms can be selected: 'newreno', 'cubic', 'hd', 'htcp', 'cdg', 'compound'.

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

However, only some of these are supported depending on the OS a host is running:

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

- Linux: cubic (default), newreno, htcp;
- Mac OS X: newreno (default)

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.

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 ‘host0’ entry in TPCONF_TCP_algos will lead to each host using its default. A ‘host1’ entry will configure 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' ]
}}
```

5) Emulated delays, loss rates and bottleneck bandwidths: TPCONF_delays specifies the emulated network 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 network 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 understands:

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

6) Selection of bottleneck AQM and buffer sizes: 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' ]
```

Note that all underscores in parameter values used in log file names are changed to hyphens to allow for easier parsing of log file names. For example, ‘fq_codel’ will become ‘fq-codel’ in the file name.

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 example for a Linux router would result in buffers of 100, 200 and 600 packets long:

```
TPCONF_buffer_sizes = [ 100, 200, 600 ]
```

7) Specifying which parameters to vary: TPCONF_vary_parameters specifies a list of parameters to vary over a series of experiments, i.e. parameters that will take on multiple values. The listed parameters must be defined in TPCONF_parameter_list (see Section V-Q). The total number of experiments carried out is the number of unique parameter combinations (multiplied by the number of TPCONF_runs if 'runs' is also specified in TPCONF_vary_parameters).

The TPCONF_vary_parameters specification also defines the order of the parameters in the log file names. While not strictly necessary, if used 'runs' should be last in the list. 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 parameters included in TPCONF_parameter_list in the example config files:

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

P. Experiment-specific variables

Some variables defined in the example config file(s) 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 (as integer) for the replies sent in an incast scenario. TPCONF_inc_periods specifies the length of a period between requests by the querier in seconds (as floating point). The following shows an example:

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

Q. Defining parameters to vary

TEACUP provides a flexible mechanism for defining what parameters are varied during experiments, what values (or ranges of values) those parameters may take, and how those parameters are then used to drive host, traffic generator and bottleneck configuration. TPCONF_parameter_list is a Python dictionary at the core of this mechanism.

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 'short 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 to set (this can be empty), where 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 parameters 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, where in each experiment a specific combination of the grouped V_variables is used. An example of this is the parameter bandwidths which uses TPCONF_bandwidths as values.

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 14.

V_delay can then 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

```
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 14: Specifying the parameters to be varied

start of the log file names (we assume the test ID prefix is the default from Section V-F):

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

R. 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 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.

VI. 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.

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 (and run_resume.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 one of the provided example config files as basis and modify it as necessary.

B. Example config

Listing 1 shows a minimal but complete config.py file. The testbed consists of three machines, 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 sent 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, network delay and loss, upstream and downstream bandwidths, AQM technique, and buffer size. There is one experiment for each combination of parameters (one run).

C. Running experiments

There are two Fabric 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:

```
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_linux_kern_router = '3.14.18-10000hz'
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_max_time_diff = 1

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', ]
```

Listing 1: Example config.py file

```
> fab run_experiment_single
```

To run a series of experiment based on the TP-CONF_vary_parameters settings 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 (stdout) and can be redirected with the usual means. The default test ID prefix (TP-CONF_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 TEACUP provides a shell script (`run.sh`) that logs the Fabric output in a <test_ID_prefix>.log file inside the <test_ID_prefix> sub directory, 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
```

TEACUP 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 TEACUP never resets either of these files – new test IDs are simply appended to the files (which are created if they don't already exist).[7]

[7]The user must manually delete or edit these files if actual experiment results are later deleted.

A run_experiment_multiple task that was interrupted part-way through may be restarted with the `resume` parameter. TEACUP will perform all experiments of the series that were not previously completed (not logged in `experiments_completed.txt`).

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 also enables redoing of previously completed experiments, by first editing them out of `experiments_completed.txt.`

If a series of experiments is interrupted by non-deterministic errors, i.e. each experiment may fail with some small probability, the `run_resume.sh` shell script can be used to ensure the whole series of experiments is completed. The script runs the experiments using the run_experiment_multiple task and uses the `resume` option to automatically restart and continue each time an experiment was not successfully completed. The script is used by executing:

```
> run_resume.sh
```

D. TEACUP variable information logged

TEACUP logs configuration information for conducted experiments in two files. In the following <test_id> is a test ID and <test_id_pfx> is the corresponding test ID prefix. For each experiment, TEACUP logs all V_ variables and their values (in alphabetical order of variable names) in a file <test_id>_config_vars.log.gz. The following is an example (with header and legend lines omitted for brevity):

```
U V_aqm: pfifo
U V_bsize: 1000
U V_delay: 100
```

Each line starts with an 'U' or an 'N' indicating whether a variable is Used or Not used. After the use indicator follows the variable name and the value the variable was set to in the particular experiment.

TEACUP also logs all varying parameters for a series of experiments in a file named <test_id_pfx>_varying_params.log.gz. The following shows a file as example (excluding the header line):

```
V_aqm aqm aqms
V_bsize bs bsizes
V_delay del delays
```

In each line, the first column is the V_ variable name, the second column is the short name used in the test ID (in the file names), and the last column is the identifier that is used in TPCONF_vary_parameters (and links to an entry in TPCONF_parameter_list).

VII. Host Control Utility Functions

This section describes a number of utility functions available as Fabric tasks. As mentioned previously, the fab utility has an option to list all available tasks:

```
> fab -l
```

A. Remote command execution

The exec_cmd task can be used to execute one command on one or more testbed 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 set of TPCONF_router and TPCONF_hosts). For example, the following command is executed on all testbed hosts:

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

B. Copying files to testbed hosts

The copy_file task can be used to copy a local file to one or more 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 set 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 parameter `method` controls the method used for copying. By default (`method='put'`) copy_file will use the Fabric put function to copy the file. However, the Fabric put function is slow. For large files setting `method='scp'` provides much better performance using the scp command. While scp is faster, it may prompt for the password if public key authentication has not been configured.

C. Installing ssh keys

The authorize_key task can be used to append the current user's public SSH 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 three 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.

D. Topology configuration

Where the testbed meets the requirements described in Section V-H, the init_topology task can be used to "move" hosts from one test subnet to the other by configuring the VLAN membership of the switch ports in conjunction with IP addresses and static routes on each host. The task uses the config.py file to get the testbed IP addresses of the hosts to be configured (from TPCONF_host_internal_ip). It also uses the values of TPCONF_topology_switch, TPCONF_topology_switch_port_prefix, and TPCONF_topology_switch_port_offset specified in config.py to reconfigure the switch. The last three parameters can also be overridden by the task's parameters `switch`, `port_prefix`, and `port_offset`. Limitations of the current implementation are described in Section V-H.

The following shows an example where we configure the hosts testhost1 and testhost2 using the task's parameters to specify the switch-relevant settings:

```
> fab -H testhost1,testhost2
init_topology:switch="switch2",
port_prefix="Gi1/0/",port_offset="5"
```

E. Initialise hosts to a specific operating system

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. Note that `os_list` can be shorter than the number of specified hosts, in which case it will be padded to the length of the number of hosts by duplicating the last entry. This allows to reboot a large number of hosts into the same OS while specifying an `os_list` with only a single entry (the desired OS).

For Linux the kernel to boot can be chosen with the parameters `linux_kern_hosts` and `linux_kern_router`. `linux_kern_hosts` specifies the kernel for normal hosts (not routers) and `linux_kern_router` specifies the kernel for routers. Please refer to the description in Section V-I2 on how to specify the kernel name.

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 the boot timeout (assuming TEACUP-compatible power controller(s) are configured). The script will then wait for `boot_timeout` seconds again for the host to come up. If the host is still unresponsive 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
```

Since version 0.9.3 TEACUP can reboot unresponsive hosts (i.e. hosts to which TEACUP cannot establish an SSH connection), if the user specifies the MAC addresses of the control interface and a power controller is configured for the hosts. The MAC address(es) are specified with the `mac_list` parameter (comma-separated list of MAC addresses, one for each host to be rebooted).

The following command shows an example of rebooting the unresposive host testhost1 (with control network interface MAC of 01:02:03:04:05:06) into Linux:

```
> fab -H testhost1 init_os:os_list="Linux",
mac_list="01:02:03:04:05:06"
```

F. Power cycling

The power_cycle task can be used to power cycle hosts, i.e. if hosts become unresponsive (assuming TEACUP-compatible power controller are configured). After the power cycle the hosts will boot the last selected OS. For example, the following command power cycles the hosts testhost1 and testhost2:

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

G. Check software installations on testbed hosts

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
```

H. Check testbed host connectivity

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
```

I. Check TEACUP config file

The check_config task can be used to check the TEACUP config file. This task will perform a number of checks and abort with an error message if it finds any errors in the config file. (This task is automatically run at the start of each experiment or series of experiments.)

```
> fab check_config
```

1) Print version information: The get_version task prints out the TEACUP version number, revision number and data, and the copyright:

```
> fab get_version
```

The output will look like:

```
TEACUP Version 0.9
Revision: 1175
Date: 2015-04-01 11:00:50 +1100 (Wed, 01
Apr 2015)
Copyright (c) 2013-2015 Centre for
Advanced Internet Architectures
Swinburne University of Technology. All
rights reserved.
```

VIII. Experiment Examples

A. Overview

Here we provide a few example scenarios of how TEACUP can be used. Each scenario involves traffic senders and receivers distributed across two subnets (subnet A and subnet B) connected by a bottleneck router. All hosts and the router are time synchronised using NTP. We have the following scenarios.

- Scenario 1: We have two TCP flows with data flowing from a source in subnet A to a destination in subnet B. We emulate different delays on the router. In this scenario the rebooting functionality of TEACUP is not used, which means the experimenter has to boot all three machines into the desired OS before the experiment is started.
- Scenario 2: As with scenario 1, with TEACUP also automatically booting the desired OS.
- Scenario 3: As with scenario 2, with TEACUP using a power controller to power cycle hosts if they don't respond after the reboot.
- Scenario 4: As with scenario 1, with each TCP flow between a different sender/receiver pair and different network path delay emulated for each flow (both flows still go through the same bottleneck).
- Scenario 5: We have three staggered bulk-transfer flows going through the router, each between a different sender/receiver pair. We use different emulated delay for each sender/receiver pair and also vary the bandwidths. We now also vary the TCP congestion control algorithm.
- Scenario 6: We have three hosts plus the router. One host in subnet A acts as web server. The two other hosts in subnet B act as clients that both use DASH-like video streaming over HTTP. We emulate different network delays and AQM mechanisms.
- Scenario 7: Investigating the incast problem. On host in subnet A queries 10 responders in subnet B. Again, we emulate different network delays, AQM mechanisms and TCP algorithms. We also vary the size of the responses.

B. Scenario 1: Two TCP flows from data sender to receiver

1) Topology: In this scenario we have two hosts: newtcp20 connected to the 172.16.10.0/24 network and newtcp27 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets. All three machines also have a second network interface that is used to control the experiment via TEACUP. newtcp20 and newtcp27 must run Linux, FreeBSD, Windows 7 or Mac OS X and newtcprt3 must run FreeBSD or Linux. newtcp20 and newtcp27 must have the traffic generator and logging tools installed as described in [7]. However, PXE booting or a multi-OS installation is not needed for this scenario.

2) Test Traffic: Two TCP bulk transfer flows are created using iperf.

3) Variable Parameters: We emulate three different delays and two different bandwidth settings – six different experiments in total. We do also define a variable parameter for AQM, but define only one AQM (default pfifo). This causes the used AQM to be logged as part of the experiment ID.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the configuration file.

Note that TEACUP configuration files can be split across multiple files (using Python's execfile()). This allows to split a large config file into multiple files for better readability. It also allows to include part of configuration files into multiple different config files, which makes it possible to reuse parts of the config that is not changed across multiple experiments. An example of a this is this config file, which is functionally identical to the above config file. Only here the main config file includes separate files to specify the testbed machines, the router setup and the traffic generated.

At the top of the configuration file we need to import the Fabric env structure and we also need to import any other Python functionality used (see Listing 2). First we need to configure the user and password used by Fabric via Fabrics env parameters. A password is not required if public key authentication is set up. We also need to tell Fabric how to execute commands on the remote machines. TEACUP uses Bourne shell (/bin/sh) by default.

The next part of the config file defines the path to the TEACUP scripts and the testbed hosts (see Listing 3). TPCONF_router is used to define the router and TPCONF_hosts is used to define the list of hosts. For each host and the router the testbed network interfaces need to be defined with TPCONF_host_internal_ip. The router obviously has two testbed network interfaces, whereas the hosts have only one.

In the next part of the configuration file we need to define some general experiment settings (see Listing 4). TEACUP_max_time_diff specifies the maximum clock offset in seconds allowed. This is a very coarse threshold as the offset estimation performed by TEACUP is not very accurate. Currently, TEACUP simply attempts to find out if the synchronisation is very bad and if yes it will abort; it does try to enforce high accuracy for the time synchronisation. TPCONF_test_id defines the name prefix for the output files for an experiment or a series of experiments. TPCONF_remote_dir specifies where on the hosts the log files are stored until they are moved to the control host running TEACUP after each experiment.

Then we define the router queues/pipes using TPCONF_router_queues (see Listing 5). Each entry of this list is a tuple. The first value is the queue number and the second value is a comma separated list of parameters. The queue numbers must be unique. Note that variable parameters must be either constants or variable names defined by the experimenter. Variables are evaluated during run-time. Variable names must start with a 'V_'. Parameter names can only contain numbers, letter (upper and lower case), underscores (_), and hyphen/minus (-). All V_variables must be defined in TPCONF_variable_list (see below).

Next we need to define the traffic generated during the experiments (see Listing 6). TEACUP implements a number of traffic generators. In this example we use iperf to generate TCP bulk transfer flows. TPCONF_traffic_gens defines the traffic generator, but here we use a temporary variable as well, which allows to have multiple traffic generator definitions and switch between them by changing the variable assigned to TPCONF_traffic_gens.

Each entry in is a 3-tuple. The first value of the tuple must be a float and is the time relative to the start of the experiment when tasks are executed. If two tasks have the same start time their start order is arbitrary. The second entry of the tuple is the task number and must be a unique integer (used as ID for the process). The last value of the tuple is a comma separated list of parameters. The first parameter of this list must be the task name. The TEACUP manual lists the task name and possible parameters. Client and server can be specified using the external/control IP addresses or host names. In this case the actual interface used is the *first* internal address (according to TPCONF_host_internal_ip). Alternatively, client and server can be specified as internal addresses, which allows to use any internal interfaces configured.

Next, we define all the parameter values used in the experiments (see Listing 7). TPCONF_duration defines the duration of the traffic. TPCONF_runs specifies the number of runs carried out for each unique combination of parameters. TPCONF_TCP_algos specifies the congestion control algorithms. Here we only use Newreno.

In this simple case all hosts use the same TCP congestion control algorithm, but TEACUP allows to specify per-host algorithms with TPCONF_host_TCP_algos. Parameter settings for TCP congestion control algorithms can be specified with TPCONF_host_TCP_algo_params (assuming parameters can be controlled with sysctl), but

```
# User and password
env.user = 'root'
env.password = 'rootpw'
# Set shell used to execute commands
env.shell = '/bin/sh -c'
```

Listing 2: Scenario 1, Fabric configuration

```
# Path to teacup scripts
TPCONF_script_path = '/home/teacup/teacup-0.8'
# DO NOT remove the following line
sys.path.append(TPCONF_script_path)
# Set debugging level (0 = no debugging info output)
TPCONF_debug_level = 0
# Host lists
TPCONF_router = ['newtcprt3', ]
TPCONF_hosts = [ 'newtcp20', 'newtcp27', ]
# Map external IPs to internal IPs
TPCONF_host_internal_ip = {
        'newtcprt3': ['172.16.10.1', '172.16.11.1'],
        'newtcp20': ['172.16.10.60'],
        'newtcp27': ['172.16.11.67'],
}
```

Listing 3: Scenario 1, general settings and host configuration

```
# Maximum allowed time difference between machines in seconds
# otherwise experiment will abort cause synchronisation problems
TPCONF_max_time_diff = 1
# Experiment name prefix used if not set on the command line
# The command line setting will overrule this config setting
now = datetime.datetime.today()
TPCONF_test_id = now.strftime("%Y%m%d-%H%M%S") + 'experiment'
# Directory to store log files on remote host
TPCONF_remote_dir = '/tmp/'
```

Listing 4: Scenario 1, test ID prefix and directory configuration

here we do not make use of that and simply use the default settings.

TPCONF_delays specifies the delay values (delay in each direction), TPCONF_loss_rates specifies the possible packet loss rates and TPCONF_bandwidths specifies the emulated bandwidths (in downstream and upstream directions). TPCONF_aqms specifies the AQM mechanism to use (here pfifo which is the default). TPCONF_buffer_sizes specifies the size of the queue. This is normally specified in packets, but it depends on the type of AQM. For example, if bfifo was used the size would need to be specified in bytes.

Finally, we need to specify which parameters will be varied and which parameters will be fixed for a series of experiments, and we also need to define how our parameter ranges above map to V_variables used for the queue setup and traffic generators and the log file names generated by TEACUP (see Listing 8).

TPCONF_parameter_list is a map of the parameters we potentially want to vary. The key of each item is the identifier that can be used in TPCONF_vary_parameters (see below). The value of each item is a 4-tuple containing the following things. First, a list of variable names. Second, a list of short names uses for the file names. For each parameter varied a string '_<short_name>_<value>' is appended to the log file names (appended to chosen prefix). Note, short names should only be letters from a–z or A–Z, do not use underscores or hyphens. Third, the list of parameters values. If there is more than one variable this must be a list of tuples, each tuple having

```
TPCONF_router_queues = [
        # Set same delay for every host
        ('1', " source='172.16.10.0/24', dest='172.16.11.0/24', delay=V_delay, "
          " loss=V_loss, rate=V_up_rate, 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_down_rate, queue_disc=V_aqm, queue_size=V_bsize "),
]
```

Listing 5: Scenario 1, router queue configuration

```
traffic_iperf = [
        ('0.0', '1', " start_iperf, client='newtcp27', server='newtcp20', port=5000, "
                 " duration=V_duration "),
        ('0.0', '2', " start_iperf, client='newtcp27', server='newtcp20', port=5001, "
                 " duration=V_duration "),
        # Or using internal addresses
        #( '0.0', '1', " start_iperf, client='172.16.11.2', server='172.16.10.2', "
        #              " port=5000, duration=V_duration " ),
        #( '0.0', '2', " start_iperf, client='172.16.11.2', server='172.16.10.2', "
        #              " port=5001, duration=V_duration " ),
]
# THIS is the traffic generator setup we will use
TPCONF_traffic_gens = traffic_iperf
```

Listing 6: Scenario 1, traffic configuration

```
# Duration in seconds of traffic
TPCONF_duration = 30
# Number of runs for each setting
TPCONF_runs = 1
# TCP congestion control algorithm used
TPCONF_TCP_algos = ['newreno', ]
# Emulated delays in ms
TPCONF_delays = [0, 25, 50]
# Emulated loss rates
TPCONF_loss_rates = [0]
# Emulated bandwidths (downstream, upstream)
TPCONF_bandwidths = [ ('8mbit', '1mbit'), ('20mbit', '1.4mbit'), ]
# AQM
TPCONF_aqms = ['pfifo', ]
# Buffer size
TPCONF_buffer_sizes = [100]
```

Listing 7: Scenario 1, experiment parameter value configuration

the same number of items as the number of variables. Fourth, an optional dictionary with additional variables, where the keys are the variable names and the values are the variable values.

The parameters that are actually varied are specified with TPCONF_vary_parameters. Only parameters listed in TPCONF_vary_parameters will appear in the name of TEACUP output files. So it can make sense to add a parameter in TPCONF_vary_parameters that only has a single value, because only then will the parameter short name and value be part of the file names.

TPCONF_variable_defaults specifies the default value for each variable, which is used for variables that are not varied. The key of each item is the parameter name. The value of each item is the default parameter value used if the variable is not varied.

5) Running Experiment: Create a directory with the above config.py and copy fabfile.py and run.sh from the

```
TPCONF_parameter_list = {
        # Vary name V_ variable file name values extra vars
        '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_down_rate', 'V_up_rate'], ['down', 'up'], TPCONF_bandwidths, {}),
}
TPCONF_variable_defaults = {
        # V_ variable value
        'V_duration' : TPCONF_duration,
        'V_delay' : TPCONF_delays[0],
        'V_loss' : TPCONF_loss_rates[0],
        'V_tcp_cc_algo' : TPCONF_TCP_algos[0],
        'V_down_rate' : TPCONF_bandwidths[0][0],
        'V_up_rate' : TPCONF_bandwidths[0][1],
        'V_aqm' : TPCONF_aqms[0],
        'V_bsize' : TPCONF_buffer_sizes[0],
}
# Specify the parameters we vary through all values, all others will be fixed
# according to TPCONF_variable_defaults
TPCONF_vary_parameters = ['delays', 'bandwidths', 'aqms', 'runs',]
```

Listing 8: Scenario 1, variable parameters configuration

TEACUP code distribution into this directory. To run the series of experiments with all parameter combinations go into the experiment directory (that contains fabfile.py and run.sh) and execute:

```
> ./run.sh
```

This will create a sub directory with the name of the test ID prefix and store all output files in this sub directory.

6) Analysing Results: TEACUP provides a range of tasks for analysing the results of a given experiment, described in an accompanying technical report [8]. Here we briefly summarise how to extract some key results.

TEACUP will create a file names <test_id_prefix>.log file in the experiment data sub directory. It will also create a file experiments_started.txt and experiments_completed.txt in the parent directory. The file experiments_started.txt contains the names of all experiments started and the file experiments_completed.txt contains the names of all experiments successfully completed. If some experiments were not completed, check the log file for errors.

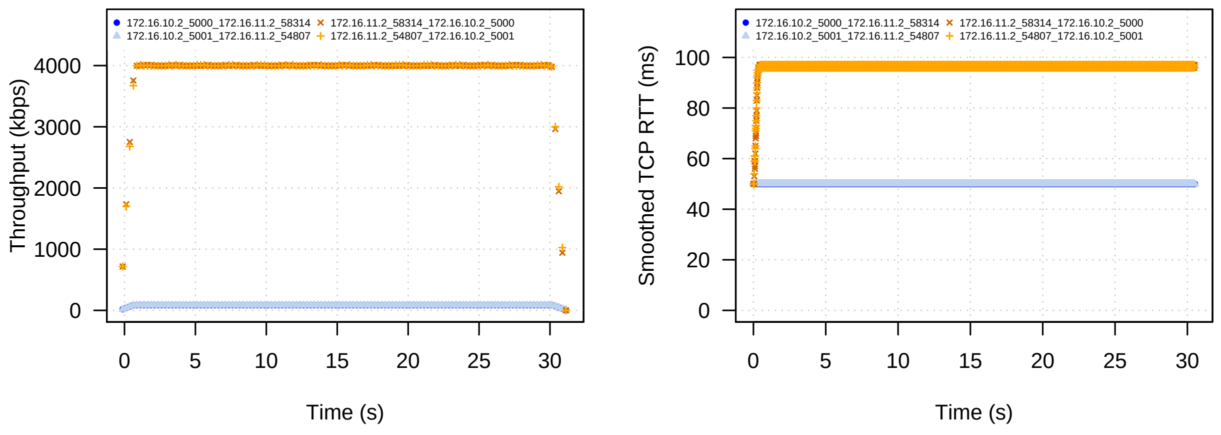
Assuming all experiments of a series completed successfully, we can now start to analyse the data. To create time series of CWND, RTT as estimated by TCP, RTT as estimated using SPP and the throughput over time for each experiment of the series, use the following command which creates intermediate files and the .pdf plot files in a sub directory named results inside the test data directory (the command needs to be executed in the directory where fabfile.py is).

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

Figure 15 shows the throughput and TCP's smoothed RTT estimates produced by TEACUP for the experiment with an emulated RTT of 50 ms, a bandwidth of 8 mbit/s downstream and 1 mbit/s upstream, and a buffer size of 100 packets. The hosts and the router ran Linux kernel 3.17.4. The throughput graph shows that both flows share the downstream link equally, while the upstream link carrying only ACKs is not fully utilised. The RTT graph shows that the total RTT reaches almost 100 ms due to the buffering (the plotted RTT estimates for the ACK streams are useless).

C. Scenario 2: Scenario 1 plus automatic booting of hosts

1) Topology: Like in scenario 1 we have two hosts: newtcp20 connected to the 172.16.10.0/24 network and newtcp27 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets.



(a) Throughput (b) RTT

Figure 15: Throughput and TCP RTT estimate measured in Scenario 1

All three machines also have a second network interface that is used to control the experiment via TEACUP. In this scenario we assume that the hosts have been installed according to [7] including PXE booting and a multi-OS installation on each host as described in the tech report.

2) Test Traffic: Like in scenario 1 two TCP bulk transfer flows are created using iperf.

3) Variable Parameters: Like in scenario 1 we emulate three different delays and two different bandwidth settings – six different experiments in total. We do also define a variable parameter for AQM, but define only one AQM (default pfifo). This causes the used AQM to be logged as part of the experiment ID.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the configuration file.

Most of the configuration is identical to the configuration used in scenario 1. The only difference is that now TEACUP will reboot the machines automatically into the specified OS, which is very useful if machines have a multi-boot setup, i.e. can run different OS. The automatic rebooting of multi-OS machines requires that machines are configured with PXE+GRUB as described in the tech report. If machines have only a single OS the reboot function is still useful to put machines into a clean state before an experiment and in this case PXE booting is not needed.

Listing 9 shows the additional configuration needed, compared to scenario 1. TPCONF_tftpboot_dir specifies the directory where TEACUP will put the files that GRUB will read (after loaded via PXE) which specify from which hard disk partition to boot from. TPCONF_host_os specifies the operating for each host and router. For Linux TEACUP allows to specify the kernel that is booted. TPCONF_linux_kern_router specifies the kernel booted on the router and TPCONF_linux_kern_hosts specifies the kernel booted on the other hosts. If TPCONF_force_reboot is not set to '0', TEACUP will only reboot a host if the currently running OS is different from the OS specified in TPCONF_host_os (Linux hosts will also be rebooted if the currently running kernel is different from the kernel specified in TPCONF_linux_kern_router or TPCONF_linux_kern_hosts). TPCONF_boot_timeout specifies the amount of time TEACUP will wait for a host to be rebooted and accessible via SSH again. If a host is not rebooted and running the desired OS within this time TEACUP will abort with an error.

Currently, by default TEACUP expects Windows on partition 1, Linux on partition 2, FreeBSD on partition 3 on the first hard disk. However, the variable TPCONF_os_partition can be used to specify the partitions in GRUB4DOS format. PXE booting of MacOS is not supported currently.

5) Run and analyse experiment: See scenario 1.

```
# Path to tftp server handling the pxe boot
# Setting this to an empty string '' means no PXE booting, and TPCONF_host_os
# and TPCONF_force_reboot are simply ignored
TPCONF_tftpboot_dir = '/tftpboot'
# Operating system config, machines that are not explicitly listed are
# left as they are (OS can be 'Linux', 'FreeBSD', 'CYGWIN', 'Darwin')
TPCONF_host_os = {
        'newtcprt3': 'Linux',
        'newtcp20': 'Linux',
        'newtcp27': 'Linux',
}
# Specify the Linux kernel to use, only used for machines running Linux
# (basically the full name without the vmlinuz-)
TPCONF_linux_kern_router = '3.17.4-vanilla-10000hz'
TPCONF_linux_kern_hosts = '3.17.4-vanilla-web10g'
# Force reboot
# If set to '1' will force a reboot of all hosts
# If set to '0' only hosts where OS is not the desired OS will be rebooted
TPCONF_force_reboot = '0'
# Time to wait for reboot in seconds (integer)
# Minimum timeout is 60 seconds
TPCONF_boot_timeout = 120
# Map OS to partition on hard disk (note the partition must be specified
# in the GRUB4DOS format, _not_ GRUB2 format)
TPCONF_os_partition = {
        'CYGWIN': '(hd0,0)',
        'Linux': '(hd0,1)',
        'FreeBSD': '(hd0,2)',
}
```

Listing 9: Scenario 2, OS and reboot configuration

D. Scenario 3: Scenario 1 plus power control

1) Topology: Like in scenario 1 we have two hosts: newtcp20 connected to the 172.16.10.0/24 network and newtcp27 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets. All three machines also have a second network interface that is used to control the experiment via TEACUP. We assume that the hosts have been installed according to [7]. Furthermore, this scenario only works with installed TEACUP-compatible power controllers that are set up to control the power of the three hosts.

2) Test Traffic: Like in scenario 1 two TCP bulk transfer flows are created using iperf.

3) Variable Parameters: Like in scenario 1 we emulate three different delays and two different bandwidth settings – six different experiments in total. We do also define a variable parameter for AQM, but define only one AQM (default pfifo). This causes the used AQM to be logged as part of the experiment ID.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the configuration file.

Most of the configuration is identical to the configuration used in scenario 2. The only different is that now TEACUP will automatically power cycle machines that to not come up within the reboot timeout. TEACUP will only power cycle machines once. If after a power cycle and reboot the machines are still unresponsive TEACUP will give up. Power cycling can only used if the machines are connected via a power controller supported by TEACUP. Currently, TEACUP supports two power controllers: IP Power 9258HP (9258HP) and Serverlink SLP-SPP1008-H (SLP-SPP1008).

Listing 10 shows the additional configuration needed, compared to scenario 1. TPCONF_do_power_cycle must be set to ‘1’ to perform power cycling. If power cycling is used TPCONF_host_power_ctrlport must define the IP address of the responsible power controller for each machine as well as the power controller's port the machine is connected to. TPCONF_power_admin_name and TPCONF_power_admin_pw must specify the admin user's name and password required to login to the power controller's web interface. TPCONF_power_ctrl_type specifies the type of power controller.

5) Run and analyse experiment: See scenario 1.

E. Scenario 4: TCP flows with same bottleneck queue but different delay

1) Topology: We now have two hosts in each subnet: newtcp20 and newtcp21 connected to the 172.16.10.0/24 network, newtcp27 and newtcp28 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets. All three machines also have a second network interface that is used to control the experiment via TEACUP. Like scenario 1 this scenario requires that hosts have the traffic generator and logging tools installed as described in [7], but PXE booting or a multi-OS installation is not needed.

2) Test Traffic: Two TCP bulk transfer flows are created using iperf. One flow is between newtcp20 and newtcp27 and the second flow is between newtcp21 and newtcp28.

3) Variable Parameters: Like in scenario 1 we emulate two different bandwidth settings. However, we setup different delays for each flow. Since each flow can have two different values, we have eight different experiments in total. We do also define a variable parameter for AQM, but define only one AQM (default pfifo). This causes the used AQM to be logged as part of the experiment ID.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the parts of the configuration file that have changed compared to scenario 1.

Our host configuration now looks as in Listing 11.

To configure different delays for each flow we also need to change the router setup (see Listing 12).

We also need to define both of the V_ variables and make sure we iterate over both in the experiments (see Listing 13).

Finally, we need to define the traffic generators to create one TCP bulk transfer flow for each host pair as shown in Listing 14.

5) Run and analyse experiment: See scenario 1.

F. Scenario 5: Partially overlapping TCP flows with different TCP CC

1) Topology: We now have three hosts in each subnet: newtcp20, newtcp21 and newtcp22 connected to the 172.16.10.0/24 network, and newtcp27, newtcp28 and newtcp29 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets. All three machines also have a second network interface that is used to control the experiment via TEACUP. Like scenario 1 this scenario requires that hosts have the traffic generator and logging tools installed as described in [7], but PXE booting or a multi-OS installation is not needed.

2) Test Traffic: Three TCP bulk transfer flows are created using iperf. One flow is between newtcp20 and newtcp27, the second flow is between newtcp21 and newtcp28, and the third flow is between newtcp22 and newtcp29. The flows do not longer start at the same time. Flow one start at the start of the experiment, while flow 2 starts 10 seconds after the first flow and flow 3 starts 10 seconds after flow 2. All flows have a duration of 30 seconds as in scenario 1.

3) Variable Parameters: Like in scenario 1 we emulate three different delays (same delays for all flows) and two different bandwidth settings. However, we now also vary the used TCP congestion control algorithm between Newreno and Cubic. This means we have 12 different experiments in total. We do also define a variable parameter for AQM, but define only one AQM (default pfifo). This causes the used AQM to be logged as part of the experiment ID.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the parts of the configuration file that have changed compared to scenario 1.

Our host configuration now looks as shown in Listing 15.

The traffic generator setup now creates three staggered flows as shown in Listing 16.

We also need to configure the different TCP congestion control algorithms and instruct TEACUP to vary this parameter (see Listing 17).

```
# If host does not come up within timeout force power cycle
# If set to '1' force power cycle if host not up within timeout
# If set to '0' never force power cycle
TPCONF_do_power_cycle = '1'
# Maps host to power controller IP (or name) and power controller port number
TPCONF_host_power_ctrlport = {
        'newtcprt3': ('10.0.0.100', '1'),
        'newtcp20': ('10.0.0.100', '2'),
        'newtcp27': ('10.0.0.100', '3'),
}
# Power controller admin user name
TPCONF_power_admin_name = 'admin'
# Power controller admin user password
TPCONF_power_admin_pw = env.password
# Type of power controller. Currently supported are only:
# IP Power 9258HP (9258HP) and Serverlink SLP-SPP1008-H (SLP-SPP1008)
TPCONF_power_ctrl_type = 'SLP-SPP1008'
```

Listing 10: Scenario 3, power controller configuration

```
TPCONF_router = ['newtcprt3', ]
TPCONF_hosts = [ 'newtcp20', 'newtcp21', 'newtcp27', 'newtcp28', ]
TPCONF_host_internal_ip = {
        'newtcprt3': ['172.16.10.1', '172.16.11.1'],
        'newtcp20': ['172.16.10.60'],
        'newtcp21': ['172.16.10.61'],
        'newtcp27': ['172.16.11.67'],
        'newtcp28': ['172.16.11.68'],
}
```

Listing 11: Scenario 4, host configuration

```
TPCONF_router_queues = [
        ('1', " source='172.16.10.60', dest='172.16.11.67', delay=V_delay, "
          " loss=V_loss, rate=V_up_rate, queue_disc=V_aqm, queue_size=V_bsize "),
        ('2', " source='172.16.11.67', dest='172.16.10.60', delay=V_delay, "
          " loss=V_loss, rate=V_down_rate, queue_disc=V_aqm, queue_size=V_bsize "),
        ('3', " source='172.16.10.61', dest='172.16.11.68', delay=V_delay2, "
          " loss=V_loss, rate=V_up_rate, queue_disc=V_aqm, queue_size=V_bsize, "
          " attach_to_queue='1' "),
        ('4', " source='172.16.11.68', dest='172.16.10.61', delay=V_delay2, "
          " loss=V_loss, rate=V_down_rate, queue_disc=V_aqm, queue_size=V_bsize, "
          " attach_to_queue='2' "),
]
```

Listing 12: Scenario 4, router queue configuration

5) Run and analyse experiment: See scenario 1.

G. Scenario 6: Two HTTP-based video streaming clients

1) Topology: We now have hosts newtcp20, newtcp21 connected to the 172.16.10.0/24 network and host newtcp27 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets. All three machines also have a second network interface that is used to control the experiment via TEACUP. Like scenario 1 this scenario requires that hosts have the traffic generator and logging tools installed as described in [7], but PXE booting or a multi-OS installation is not needed.

2) Test Traffic: In this scenario we simulate DASH-like HTTP video streaming. Host newtcp27 runs a web

```
TPCONF_delays = [5, 50]
TPCONF_delays2 = [5, 50]
TPCONF_parameter_list = {
        # Vary name V_ variable file name values extra vars
        'delays' : (['V_delay'], ['del1'], TPCONF_delays, {}),
        'delays2' : (['V_delay2'], ['del2'], TPCONF_delays2, {}),
        '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_down_rate', 'V_up_rate'], ['down', 'up'], TPCONF_bandwidths, {}),
}
TPCONF_variable_defaults = {
        # V_ variable value
        'V_duration' : TPCONF_duration,
        'V_delay' : TPCONF_delays[0],
        'V_delay2' : TPCONF_delays2[0],
        'V_loss' : TPCONF_loss_rates[0],
        'V_tcp_cc_algo' : TPCONF_TCP_algos[0],
        'V_down_rate' : TPCONF_bandwidths[0][0],
        'V_up_rate' : TPCONF_bandwidths[0][1],
        'V_aqm' : TPCONF_aqms[0],
        'V_bsize' : TPCONF_buffer_sizes[0],
}
TPCONF_vary_parameters = ['delays', 'delays2', 'bandwidths', 'aqms', 'runs',]
```

Listing 13: Scenario 4, experiment parameter configuration

```
traffic_iperf = [
        ('0.0', '1', " start_iperf, client='newtcp27', server='newtcp20', port=5000, "
                 " duration=V_duration "),
        ('0.0', '2', " start_iperf, client='newtcp28', server='newtcp21', port=5001, "
                 " duration=V_duration "),
]
TPCONF_traffic_gens = traffic_iperf
```

Listing 14: Scenario 4, traffic configuration

server. Host newtcp20 and newtcp21 are clients and use httperf to emulate DASH-like streaming – there is one DASH-like flow between each client and the server. In this scenario we have fixed the video rates and on/off cycle times, but the rate and cycle time differs for both streams.

3) Variable Parameters: Like in scenario 1 we emulate three different delays (same delays for all flows) and two different bandwidth settings. We now also vary the AQM mechanism used on the router between FIFO, CoDel and FQ CoDel. This means we have 18 experiments in total.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the parts of the configuration file that have changed compared to scenario 1.

Our host configuration now looks as in Listing 18.

The traffic generator setup now starts a web server and generates fake streaming content on newtcp27 before starting the DASH-like flows as shown in Listing 19.

Finally, we need to configure the different AQM mechanisms used (see Listing 20). TPCONF_vary_parameters already included the 'aqms' parameters in scenario 1, so TPCONF_parameter_list and TPCONF_variable_defaults look like in scenario 1. We only have to change TPCONF_aqms.

5) Run and analyse experiment: See scenario 1.

```
TPCONF_router = ['newtcprt3', ]
TPCONF_hosts = [ 'newtcp20', 'newtcp21', 'newtcp22', 'newtcp27', 'newtcp28', 'newtcp29', ]
TPCONF_host_internal_ip = {
        'newtcprt3': ['172.16.10.1', '172.16.11.1'],
        'newtcp20': ['172.16.10.60'],
        'newtcp21': ['172.16.10.61'],
        'newtcp22': ['172.16.10.62'],
        'newtcp27': ['172.16.11.67'],
        'newtcp28': ['172.16.11.68'],
        'newtcp29': ['172.16.11.69'],
}
```

Listing 15: Scenario 5, host configuration

```
traffic_iperf = [
        ('0.0', '1', " start_iperf, client='newtcp27', server='newtcp20', port=5000, "
                   " duration=V_duration "),
        ('10.0', '2', " start_iperf, client='newtcp28', server='newtcp21', port=5001, "
                   " duration=V_duration "),
        ('20.0', '3', " start_iperf, client='newtcp29', server='newtcp22', port=5002, "
                   " duration=V_duration "),
]
TPCONF_traffic_gens = traffic_iperf
```

Listing 16: Scenario 5, traffic configuration

H. Scenario 7: Incast problem scenario

1) Topology: We now have the host newtcp20 connected to the 172.16.10.0/24 network and hosts newtcp22–30 connected to the 172.16.11.0/24 network. The machine newtcprt3 connects the two experiment subnets. All three machines also have a second network interface that is used to control the experiment via TEACUP. Like scenario 1 this scenario requires that hosts have the traffic generator and logging tools installed as described in [7], but PXE booting or a multi-OS installation is not needed.

2) Test Traffic: In this scenario we emulate traffic to investigate the incast problem. The host newtcp20 is the querier and in regular intervals sends simultaneous queries to hosts newtcp21, newtcp22, newtcp23, newtcp24, newtcp25, newtcp26, newtcp27, newtcp28, newtcp29 and newtcp30 which then respond simultaneously. The querier uses httperf to send the queries to web servers running on all of the responders.

3) Variable Parameters: Like in scenario 6 we emulate three different delays (same delays for all flows), two different bandwidth settings, and three different AQM mechanism (FIFO, CoDel and FQ CoDel). We now also vary the size of the response between six different values. This means we have 108 experiments in total.

4) TEACUP Config File: You can download the configuration file from [25]. To use the configuration rename it to config.py. In the following we explain the parts of the configuration file that have changed compared to scenario 6.

Our host configuration now looks as in Listing 21.

The traffic generator setup now starts a web server and generates fake streaming content on each responder. Then after a delay of one second it starts the querier. Listing 22 shows the configuration. Setting up each server individually requires a large number of entries in the traffic configuration. Since version 0.9 TEACUP supports the start_httperf_incast_n traffic generator that will setup the n servers and the querier with a single entry.

Next, we need to configure the value ranges for response sizes and the time period between sending queries (see Listing 23).

Finally, we need to configure TEACUP to make sure the V_ variables used in the traffic generator setup are defined and we vary the response sizes (see Listing 24).

5) Run and analyse experiment: See scenario 1.

```
TPCONF_TCP_algos = ['newreno', 'cubic', ]
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_down_rate', 'V_up_rate'], ['down', 'up'], TPCONF_bandwidths, {}),
}
TPCONF_variable_defaults = {
        'V_duration' : TPCONF_duration,
        'V_delay' : TPCONF_delays[0],
        'V_loss' : TPCONF_loss_rates[0],
        'V_tcp_cc_algo' : TPCONF_TCP_algos[0],
        'V_down_rate' : TPCONF_bandwidths[0][0],
        'V_up_rate' : TPCONF_bandwidths[0][1],
        'V_aqm' : TPCONF_aqms[0],
        'V_bsize' : TPCONF_buffer_sizes[0],
}
TPCONF_vary_parameters = ['tcpalgos', 'delays', 'bandwidths', 'aqms', 'runs',]
```

Listing 17: Scenario 5, variable parameter configuration

```
TPCONF_router = ['newtcprt3', ]
TPCONF_hosts = [ 'newtcp20', 'newtcp21', 'newtcp27', ]
TPCONF_host_internal_ip = {
        'newtcprt3': ['172.16.10.1', '172.16.11.1'],
        'newtcp20': ['172.16.10.60'],
        'newtcp21': ['172.16.10.61'],
        'newtcp27': ['172.16.11.67'],
}
```

Listing 18: Scenario 6, host configuration

IX. Extending TEACUP Functionality

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 different sections, one for each OS (FreeBSD, Linux, Windows/Cygwin, Mac OS X). 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 modifying `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 (if any).

C. New traffic generator

Adding a new traffic generator requires adding 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 allows having the host on which the generator is started as explicit parameter (and not as Fabric hosts parameter) and having multiple traffic generators that actually use the same underlying tool (for example this is the case

```
traffic_dash = [
        # Start server and create content (server must be started first)
        ('0.0', '1', " start_http_server, server='newtcp27', port=80 "),
        ('0.0', '2', " create_http_dash_content, server='newtcp27', duration=2*V_duration, "
                 " rates='500, 1000', cycles='5, 10' "),
        # Create DASH-like flows
        ('0.5', '3', " start_httperf_dash, client='newtcp20', server='newtcp27', port=80, "
                 " duration=V_duration, rate=500, cycle=5, prefetch=2.0, "
                 " prefetch_timeout=2.0 "),
        ('0.5', '4', " start_httperf_dash, client='newtcp20', server='newtcp27', port=80, "
                 " duration=V_duration, rate=1000, cycle=10, prefetch=2.0, "
                 " prefetch_timeout=2.0 "),
]
TPCONF_traffic_gens = traffic_dash
```

Listing 19: Scenario 6, traffic configuration

```
TPCONF_aqms = ['pfifo', 'codel', 'fq_codel', ]
TPCONF_vary_parameters = ['delays', 'bandwidths', 'aqms', 'runs',]
```

Listing 20: Scenario 6, parameter configuration

```
TPCONF_router = ['newtcprt3', ]
TPCONF_hosts = [
        'newtcp20', 'newtcp21', 'newtcp22', 'newtcp23', 'newtcp24',
    'newtcp25', 'newtcp26', 'newtcp27', 'newtcp28', 'newtcp29', 'newtcp30',
]
TPCONF_host_internal_ip = {
        'newtcprt3': ['172.16.10.1', '172.16.11.1'],
        'newtcp20': ['172.16.10.60'], # querier
        'newtcp21': ['172.16.11.61'], # responders...
        'newtcp22': ['172.16.11.62'],
        'newtcp23': ['172.16.11.63'],
        'newtcp24': ['172.16.11.64'],
        'newtcp25': ['172.16.11.65'],
        'newtcp26': ['172.16.11.66'],
        'newtcp27': ['172.16.11.67'],
        'newtcp28': ['172.16.11.68'],
        'newtcp29': ['172.16.11.69'],
        'newtcp30': ['172.16.11.70'],
}
```

Listing 21: Scenario 7, host configuration

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()`. This ensures 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

```
traffic_incast = [
        # Start servers and create contents (server must be started first)
        ('0.0', '1', " start_http_server, server='newtcp21', port=80 "),
        ('0.0', '2', " start_http_server, server='newtcp22', port=80 "),
        ('0.0', '3', " start_http_server, server='newtcp23', port=80 "),
        ('0.0', '4', " start_http_server, server='newtcp24', port=80 "),
        ('0.0', '5', " start_http_server, server='newtcp25', port=80 "),
        ('0.0', '6', " start_http_server, server='newtcp26', port=80 "),
        ('0.0', '7', " start_http_server, server='newtcp27', port=80 "),
        ('0.0', '8', " start_http_server, server='newtcp28', port=80 "),
        ('0.0', '9', " start_http_server, server='newtcp29', port=80 "),
        ('0.0', '10', " start_http_server, server='newtcp30', port=80 "),
        ('0.0', '11', " create_http_incast_content, server='newtcp21', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '12', " create_http_incast_content, server='newtcp22', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '13', " create_http_incast_content, server='newtcp23', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '14', " create_http_incast_content, server='newtcp24', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '15', " create_http_incast_content, server='newtcp25', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '16', " create_http_incast_content, server='newtcp26', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '17', " create_http_incast_content, server='newtcp27', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '18', " create_http_incast_content, server='newtcp28', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '19', " create_http_incast_content, server='newtcp29', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        ('0.0', '20', " create_http_incast_content, server='newtcp30', duration=2*V_duration, "
                  " sizes=V_inc_content_sizes_str "),
        # Start querier
        ('1.0', '30', " start_httperf_incast, client='newtcp20', "
          " servers='newtcp21:80,newtcp22:80,newtcp23:80,newtcp24:80,newtcp25:80,newtcp26:80, "
      " newtcp27:80,newtcp28:80,newtcp29:80,newtcp30:80', "
      " duration=V_duration, period=V_inc_period, response_size=V_inc_size"),
]
TPCONF_traffic_gens = traffic_incast
```

Listing 22: Scenario 7, traffic configuration

```
TPCONF_inc_content_sizes = [8, 16, 32, 64, 128, 256]
TPCONF_inc_content_sizes_str = ','.join( str(x) for x in TPCONF_inc_content_sizes)
TPCONF_inc_periods = [10]
```

Listing 23: Scenario 7, incast content parameter configuration

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

```
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_down_rate', 'V_up_rate'], ['down', 'up'], TPCONF_bandwidths, {}),
        'incast_periods': (['V_inc_period'], ['incper'], TPCONF_inc_periods, {}),
        'incast_sizes' : (['V_inc_size'], ['incsz'], TPCONF_inc_content_sizes,{}),
}
TPCONF_variable_defaults = {
        'V_duration' : TPCONF_duration,
        'V_delay' : TPCONF_delays[0],
        'V_loss' : TPCONF_loss_rates[0],
        'V_tcp_cc_algo' : TPCONF_TCP_algos[0],
        'V_down_rate' : TPCONF_bandwidths[0][0],
        'V_up_rate' : TPCONF_bandwidths[0][1],
        'V_aqm' : TPCONF_aqms[0],
        'V_bsize' : TPCONF_buffer_sizes[0],
        'V_inc_period' : TPCONF_inc_periods[0],
        'V_inc_size' : TPCONF_inc_content_sizes[0],
        'V_inc_content_sizes_str': TPCONF_inc_content_sizes_str,
}
TPCONF_vary_parameters = ['incast_sizes', 'delays', 'bandwidths', 'aqms', 'runs',]
```

Listing 24: Scenario 7, variable parameter configuration

be called from `analysis.py:analyse_all()` via Fabrics `execute()` function. The new task should implement the common parameters test_id, out_dir, pdf_dir, out_name, replot_only, source_filter, min_values, etime, stime, ymin, ymax (see existing analyse tasks as examples). The new task must be added to the imports in `fabfile.py`.

X. Known Issues

During the host setup phase TEACUP enables and disables NICs. On Windows the enable and disable NIC commands have permanent effect. If TEACUP is interrupted or aborts between a disable and enable command, the NIC will stay disabled. TEACUP will automatically enable all testbed NICs on Windows prior to each experiment, however in the unlikely event that the previous aborted NIC configuration left the NIC in an inconsistent state, it may be necessary to reconfigure the NIC manually.

TEACUP logs all output from traffic generators, such as iperf or httperf. Some of the tools used only generate output after they completed. If an experiment ends *before* a tool completed its task, the resulting log file may be empty. Possibly this issue could be mitigated by turning the stdout and stderr buffering off for these tools in future versions.

XI. 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 versions prior to 1.0 were developed as part of a project funded by Cisco Systems and titled "Study in TCP Congestion Control Performance In A Data Centre". This was a collaborative effort between Swinburne University of Technology's Center for Advanced Internet Archectures and Fred Baker of Cisco Systems. These may be found at http://caia.swin.edu.au/tools/teacup/downloads.html.

Starting with TEACUP v1.0, development will be community supported and publicly hosted at http://sourceforge.net/projects/teacup/.

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] S. Zander, G. Armitage, "TEACUP v0.9 – A System for Automated TCP Testbed Experiments," Centre for Advanced Internet Architectures, Swinburne University of Technology, Tech. Rep. 150414A, 2015. [Online]. Available: http://caia.swin.edu.au/reports/150414A/CAIA-TR-150414A.pdf

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

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

[7] S. Zander, G. Armitage, "CAIA Testbed for TCP Experiments Version 2," Centre for Advanced Internet Architectures, Swinburne University of Technology, Tech. Rep. 150210C, 2015. [Online]. Available: http://caia.swin.edu.au/reports/150210C/CAIA-TR-150210C.pdf

[8] ——, "TEACUP v1.0 – Data Analysis Functions," Centre for Advanced Internet Architectures, Swinburne University of Technology, Tech. Rep. 150529B, 2015. [Online]. Available: http://caia.swin.edu.au/reports/150529B/CAIA-TR-150529B.pdf

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

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

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

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

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

[14] 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

[15] "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/

[16] 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

[17] Wikipedia, "DTrace," http://en.wikipedia.org/w/index.php?title=DTrace&oldid=637195163.

[18] 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

[19] 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

[20] "netfilter – firewalling, NAT, and packet mangling for Linux." [Online]. Available: http://www.netfilter.org/

[21] J. Gettys, "Best practices for benchmarking bufferbloat." [Online]. Available: http://www.bufferbloat.net/projects/codel/wiki/Best_practices_for_benc%hmarking_Codel_and_FQ_Codel

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

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

[24] C. Javier, "Broadband Internet Traffic Simulation & Synthesis (BITSS)," http://caia.swin.edu.au/bitss/.

[25] S. Zander, "TCP Experiment Automation Controlled Using Python (TEACUP) – Usage Examples," 2014. [Online]. Available: http://caia.swin.edu.au/tools/teacup/usageexamples.html