# fastping documentation

Pellegrino Casoria, Danilo Cicalese, Diana Joumblatt, Dario Rossi

May 14th, 2014

## Contents

**1 Getting started** **2**
1.1 Description . . . 2
1.2 Fastping use cases . . . 2
1.3 Fastping output . . . 3
1.4 Quick start . . . 4
1.5 mPlane support . . . 4
1.6 mPlane proxy interface . . . 4
1.7 mPlane custom registry . . . 4
1.8 Official version . . . 5
1.9 HOWTO.shell . . . 5
1.10 HOWTO.mplane . . . 6

**A System architecture** **9**

**B Top-Hat project** **9**
B.1 Lifecycle support . . . 9
B.2 System architecture . . . 10
B.2.1 User Interface . . . 10
B.2.2 Stored informations . . . 11
B.2.3 Interconnected infrastructures . . . 11
B.3 Background measurements . . . 11

**C The fastPing tool** **12**
C.1 Operational overview . . . 12
C.1.1 Measurement interval size . . . 13
C.2 Tool architecture . . . 14
C.2.1 The timing module . . . 15
C.2.2 The probing module . . . 16
C.2.3 The listener module . . . 16
C.2.4 The statistics module . . . 16
C.2.5 The inhibition module . . . 17
C.2.6 The deployment module . . . 17
C.2.7 The uploader module . . . 17

C.3 The ICMP request . . . . . 18
C.4 The target list . . . . . 19
C.5 Getting online statistics . . . . . 19
C.5.1 High-level statistics . . . . . 20
C.5.2 Low-level statistics . . . . . 21
C.5.3 Queuing delay distribution . . . . . 21
C.5.4 Raw statistics . . . . . 21
C.5.5 Results aggregation . . . . . 22
C.5.6 Memory occupation . . . . . 22
C.6 Knuth incremental algorithm (online variance) . . . . . 22
C.7 P-Square algorithm (online percentiles) . . . . . 23

**D Scalability** **24**

**E Tool infrastructure technical details** **24**
E.1 The pinger module . . . . . 26
E.2 The listener module . . . . . 27
E.3 The UPLOADER module . . . . . 28

**References** **28**

# 1 Getting started

Fastping, developed at ENST by the bunch of people above[1], is a fast ICMP scanner for TopHat/TDMI, a dedicated measurement infrastructure running over PlanetLab. If you ever received an ICMP packet containing an `http://ow.ly/wPjH6` URL, which redirects to the tool page on mPlane website `https://www.ict-mplane.eu/public/fastping`, this means that you have been a target of an academic experiment with the fastping software, that this documentation explains in details.

## 1.1 Description

"Fast" in fastping means that 50k hosts can be probed in about 5 seconds (a more detailed performance analysis is reported in later sections of this document). Scalability of a single proble is obtained in user-space (as opposite to the zmap sofware that requires root privileges), with a non-blocking multi-thread design (that allows to significantly exceed nMap Scripting Engine performance, but of course not as much as zmap).

Additionally, leveraging the TopHat/TDMI infrastructure, fastping couples the ability to scan a large number of hosts during small time windows, to the availability of a large number of spatially disperse probes – up to 1000 PlanetLab/OneLab nodes, of which typically around 300 are available at any time.

## 1.2 Fastping use cases

These temporal and spatial scalability properties can be leveraged for instance, to understand Internet properties such as (but not limited to):

[1] the last one of which has clearly written not a single line of code, so all credits should go to the former

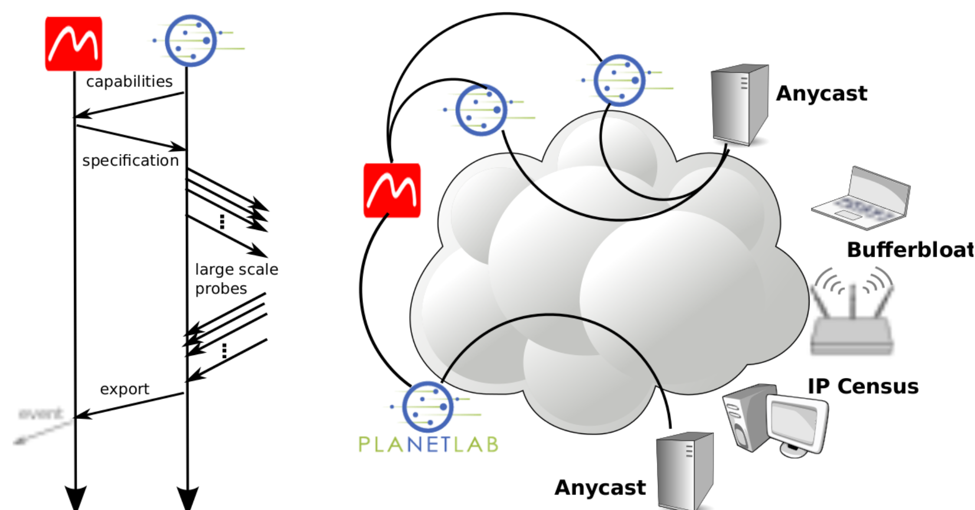


Figure 1: Fastping synoptic

- anycast detection (when all probes target the same target during the same window, but change target over time)
- bufferbloat evaluation (when each probe tracks disjoint targets, but keep the same target continuously over time)
- Internet census (when each probe tracks disjoint targets, changing target over time)

In the current status, the appendix of this document makes bufferbloat as an example to explain the whole architecture.

## 1.3 Fastping output

Fastping already perform a fair amount of statistical pre-processesing, providing output at different granularities. Namely, from the most coarse to the most fine-grained:

- (txt) experiment summary
- (cvs) per-probe statistics summary (e.g., CDF of relevant metrics as RTT delay, RTT variation, or TTL variation)
- (cvs) per-probe per-host statistics (e.g., quantiles of relevant merics as RTT delay, RTT variation, or TTL variation)
- (cvs) raw measurement

The above results can be stored locally at a probe (useful for testing/local use), or uploaded to an repository via FTP (useful to centralize data collection from a PlanetLab experiment).

## 1.4 Quick start

Fastping is a python script and

- can be used as a standalone shell tool (its usage is thus relatively simply explained by the manpage)
- can be used as a TopHat/TDMI component (out of scope of this page)
- can be queried as a mPlane probe (i.e., receive and parse mPlane specification)

Example of use of the tool as a standalone shell tool and as mPlane probe can be found in the software package (HOWTO.shell and HOWTO.mplane in the main directory respectively) as well as in later section of this document.

## 1.5 mPlane support

Fastping was entirely developed during the mPlane project. So thanks, mPlane!

- As of May 15th 201, only the mPlane client/server probe and the repository are implemented.
- The implementation of a supervisor for all fastping probes (i.e., PlanetLab and beyond) is a planned ongoing effort

## 1.6 mPlane proxy interface

The mPlane Fastping interface offers the ability to specify the set of target IP address ranges.

- (simplest form) each probe can be queried individually, i.e., specifying the target for each probe in mPlane terms, and will upload the results on the specified server
- (work in progress) measurement specification can be addressed to a dedicated mPlane supervisor, handling the TopHat/TDMI probes, that will dispatch measurement specification to all Fastping probes that registered to the supervisor

## 1.7 mPlane custom registry

To harness the full power of fastping, a number of parameters have clearly been defined in the mPlane registry. While most of the parameters are trivial, one is worth discussing at length.

To achieve efficient operation, it is imperative for fastping to receive compact specification of target addresses ranges. However, the current mPlane registry only support a `destination.ipv4` type specifying a single target address – which clearly clashes with the ability of probing at large scale.

To circumvent with this current mPlane limit, the fastmPlane implementation employs a clever trick that is both compatible with the current mPlane specifications, and that will possibly be entirely supported by future releases – since the necessity to specify list of primitive types will be possible shared among multiple components, and thus supported by mPlane.

The fastping custom registry specify thus a `destinations.ipv4range` type whereby the `s` implies that a plurality of IPv4 ranges, expressed as `A.B.C.D/N` are expected. Notice that since a single address can be expressed as a /32 range, this effectively means that it is possible to practicaly “mix” addresses and ranges, while maintaining plurality of a single type!

## 1.8 Official version

Follow the ICMP packet :)

## 1.9 HOWTO.shell

```
Instructions


1. Hello fastping
--------------------------------------

to run a single experiments targeting google DNS server 8.8.8.8
   pinging every 100ms during 1 second uploading all results to
   a local anonymous FTP:

        sudo ./fastpingBash/FastpingBash.py -t 8.8.8.8 -d 1 -f
   0.1 -n 1 -R -C -S -Q -U 127.0.0.1 anonymous '' 21 up False

The same but getting local results (archive/hostname_unixtime.rw
   , stats/hostname_unixtime.st, .... customizable path for
   output, see point 4.)

        sudo ./fastpingBash/FastpingBash.py -t 8.8.8.8 -d 1 -f
   0.1 -n 1 -R -C -S -Q



2. More serious experiment
--------------------------------------

to run 10 experiments to ENST and google DNS /16 subnets pinging
    every 100ms for 24 hours (=288 blocs of 5 minutes),
   uploading only statistics every 5minutes to a local anonymous
    FTP:

    sudo ./fastpingBash/FastpingBash.py -t T 137.194.0.0/24
   8.8.0.0/16 -d 300 -f 0.1 -1 -n 288 -S -U 127.0.0.1 anonymous
   '' 21 up False

or edit destination.txt so that it reads:

        137.194.0.0/24
        8.8.0.0/16

then run:
```

```
        sudo ./fastpingBash/FastpingBash.py -t T destination.txt
    -d 300 -f 0.1 -1 -n 288 -S -U 127.0.0.1 anonymous '' 21 up
   False




3. Even more serious experiment
--------------------------------------

as above, reading destination list from destination.txt file (
   for very large detination sets)

        sudo ./fastpingBash/FastpingBash.py -T destination.txt -
   d 300 -f 0.1 -1 -n 288 -S -U 127.0.0.1 anonymous '' 21 up
   False



4. Even even more serious experiments
--------------------------------------

Read the helper for full customizability, helper:

        sudo ./fastpingBash/FastpingBash.py -h
```

## 1.10 HOWTO.mplane

```
Instructions


1. Installation
----------------------------------------
Install mPlane required tools
        sudo apt-get install python3-yaml
        sudo apt-get install python3-tornado

Install fastPing required python2

get the fastPing code:
        svn co https://svn.ict-mplane.eu/svn/public/software/
   fastping-d22

2. Launching the probe
----------------------------------------
Start a server on your local host, in the *nix shell:
        sudo python3 -m mplane.fastping -4 127.0.0.1
```

```
To launch a client, in the *nix shell:
        python3 -m mplane.client

The user interacts with client, in the client shell:
        connect http://127.0.0.1:8888


If everything is OK, you should see:
    new client: http://127.0.0.1:8888 http://127.0.0.1:8888/
   capability
    getting capabilities from http://127.0.0.1:8888/capability
    get_mplane_reply http://127.0.0.1:8888/capability/5
   dcb7b049338dc87b842eef245a67019 200 Content-Type application/
   x-mplane+json
    parsing json
    got message:
    capability: measure
    label: fastPing
    parameters:
    curr_dir: '*'
    cycleduration.s: '*'
    destinations.ip4: '*'
    ftp.ip4: '*'
    ftp.port: '*'
    is_pasv: '*'
    numberCycle: '*'
    password: '*'
    period.s: '*'
    source.ip4: 127.0.0.1
    user: '*'
    results:
    - listFile
    token: 5dcb7b049338dc87b842eef245a67019
    when: now ... future / 1s

    adding <capability: measure (fastPing) when now ... future /
    1s token 5dcb7b04 schema 2b477234 p/m/r 11/0/1>



3. Launching an experiment
----------------------------------------

To start a 10 seconds experiment, single shot with 8.8.8.8 as
   destination, expecting results to be uploaded to a (local)
   FTP server, type in the client shell:
```

```
        when now + 1s / 10s
        set numberCycle 1
        set password ''
        set is_pasv False
        set ftp.port 21
        set ftp.ip4 127.0.0.1
        set destinations.ip4 8.8.8.8
        set curr_dir up
        set cycleduration.s 10
        set period.s 1
        set user anonymous
        runcap 0


When the experiment is over, for see the result:
        showmeas

The fastping probes supports the setup of more complex
   experiments (by using a custom registry file, which is not
   entirely supported
 by the officiel mPlane syntax). This can be triggered by
   writing the same command and changing the "destinations", as
   above:

- specify multiple hosts
        set destinations.ip4 8.8.8.8 8.8.8.4

- specify IPv4 ranges:
        set destinations.ip4 8.8.0.0/24

- specify list of IPv4 ranges:
        set destinations.ip4 8.8.0.0/24 137.194.0.0/24

- mixing everything above
        set destinations.ip4 8.8.0.0/24 8.8.8.8 137.194.0.0/24

This could be supported by mplane as either "string" (loosing
   the control at 0 complexity) or as "plurals" (keeping the
   control at more complexity)

4. Getting access to measurement results
----------------------------------------

Through FTP, CVS data (columns have headers with self-explaining
    meaning) are accessible that report data as described in D2
   .2
```

# A System architecture

Based on the previous section findings, we build a distributed architecture for Internet bufferbloat scanning. We aim at simple, reusable architecture, that can e.g., perform high-frequency sampling of large target host sets for extended durations (e.g., sending 1 packet/second to 106 targets for a week), or Internet-scale low-frequency sampling (e.g., a periodic scan of a significant fraction of Internet hosts every hour).

For this reason, we developed a simple architecture built over TopHat (9), a project initiated by UPMC during the OneLab project. In the following, we'll first describe TopHat architecture in general terms. Later, we'll focus on a more detailed description of the module we implemented in this work.

The overall system is composed by three different entities, as shown in fig. **??**:

- **agents:** the PlanetLab nodes where the TopHat system with our tool modules runs;
- **targets:** the hosts we intend to monitor;
- **repository:** the server where outputs are saved at the end of each measurement.

The agents perform the required measurements toward a specific target set and send results to the system repository; a single target can be associated to a single agent or also to multiple agent, to allow cross-validation and issue detection. Furthermore, because all the agents refer to the same repository, some expedients are needed to avoid simultaneous uploads which could reduce the system performance.

# B Top-Hat project

PlanetLab is a group of computers available as a testbed for computer networking and distributed systems research and used - rather than a simulation or emulation environment - to host experimental applications in realistic conditions over the public Internet. Such applications performances are affected by the underlying topology and its evolution: TopHat aims at developing platforms to actively probe the Internet topology, and supply live and historic measurements to the community.

TopHat uses a centralized server architecture much like other monitoring systems deployed on PlanetLab. In this way, the architecture is simple and robust at the same time, allowing to uncover bottlenecks through system logs. It relies on TDMI (TopHat Dedicated Measurement Infrastructure), which periodically makes snapshots of the overlay topology and complements its own measurements with data originating from several interconnected platforms able to provide solid and specialized measurements to the research community. This interconnection is an instance of the larger effort, pioneered by the OneLab experimental facility (26), to federate previously independent testbeds and measurement systems in order to provide a diverse global scale environment for Future Internet research.

## B.1 Lifecycle support

While several topology information services have been proposed for PlanetLab, the TopHat system is designed to support the entire lifecycle of an experiment, from setup, through run time, to retrospective analysis.

During the setup phase, it assists PlanetLab users in choosing the nodes on which the experiment will be deployed, allowing them to base decisions on measured characteristics of the network as seen from each node (e.g., location, traceroute hops, delay, bandwidths).

At run time, it provides live information on evolving topology - due to network anomalies, such as path failures, bottlenecks, and other sources of dynamism. Thus, it supports adaptive applications (e.g., a P2P application which adapt its overlay as a function of changing delays and available bandwidth) and experiment control (e.g., restart an experiment if certain paths have changed), providing measurements via a simple query interface and through the use of callbacks.

The measurement data collected by the system are archived, and are thus available for retrospective analysis of an experiment, as well as being available for the community.

TopHat also offers a log service - to help service monitoring and long term system engineering - and a collection of visualization tools that a researcher can use to obtain graphical representations of his experimental data.

## B.2 System architecture

A general TopHat architecture is shown in fig. **??**. Users access TopHat either through the web interface or at the command line (via the XMLRPC API). Applications use the XML-RPC API. The measurements originate either from TDMI or from the interconnected measurement systems via gateways.

TopHat's own measurement agents - bottom left - are deployed within a slice on PlanetLab nodes and represent the TopHat Dedicated Measurement Infrastructure (TDMI). It supplies measurements when no other system can do so. They are modular daemons that consist of wrappers around common measurement tools and basic services (e.g., upload) or also improved measurement mechanisms. The tools are invoked by dynamically loadable modules that periodically start a set of measurements (sec. B.3) to provide an XML-RPC interface to the agents (sec. B.2.1).

Other measurement infrastructure - bottom right - are interconnected to TopHat via gateways (sec. B.2.3). These last ones authenticate themselves to the external measurement infrastructures and ensure the exchange of data with TopHat. They translate the requests originating from TopHat into platform-specific requests and wrap the results in a format that TopHat can understand.

Mediating between the user and application requests and the measurement infrastructures is the core system. The core system dispatches requests and measurements to the task manager. Finally, data are stored in the storage subsystem.

### B.2.1 User Interface

TopHat offers measurements on demand through a simple query interface. The user can specify a class of measurement - such as traceroute, latency, available bandwidth, or topological distance at the IP or AS level - and TopHat itself will select the specific tool for that class: for example, when asked for a traceroute, TopHat would normally select our own team's Paris Traceroute tool (27) because of its ability to avoid many of the measurement artefacts that the standard traceroute tool encounters in the presence of load-balancing routers. Of course, the user can request a different tool - among the available ones - if he wants to.

TopHat provides two main interfaces:

- **XML-RPC API:** allows an application or experimenter (using a command-line interface) to interrogate the system;
- **WEB INTERFACE:** provides greater ergonomy for many tasks.

Typically, the API is used by the applications to perform measurements when it's running or to react to changes in the underlying network; the web interface is instead more convenient during experiment setup for such tasks as node selection.

The module we developed does not actually offer a user interface to perform on-demand measurements. Future efforts will focus on creating a mechanism not only to send ICMP echo requests to a specific set of destinations, but also to easily retrieve stored information about targets queuing delays.

### B.2.2 Stored informations

The TopHat database records measurements as well as metadata such as agent system logs. The reason is that certain types of measurements (e.g, traceroute) typically don't change much from one measurement to the next. Thus, TopHat reduces the load on its database in such circumstances by avoiding to rewrite the entire measurement and only writing the differences and new timestamps.

In addition, both the core system and the agents maintain caches to avoid the transmission and processing of redundant information: the task manager only asks the agents for a new on-demand measurement when it is unable to fulfil the request from its own cache. Finally, TopHat also stores information such as its own system logs, the dataset of IPs it is currently probing, and policy-related information (e.g., blacklists).

To limit the memory occupation of our module, we decided to not implement any kind of caching mechanism on the TDMI agents: all the output files are as soon as possible copied onto a specific server and deleted from the agent local memory. We'll discuss the module memory occupation in sec. C.5.6.

### B.2.3 Interconnected infrastructures

As we stated before, TopHat cooperates with several interconnected platforms which provide different kind of specific measurements. Between them we can find infrastructures such as:

- DIMES (28): its web service interface provides access to IP level traces and AS information;
- SONoMA (29): offers medium-precision resolution (tens of microseconds) measurements through a web service interface;
- ETOMIC (30): offers high-precision resolution (tens of nanoseconds) delay measurements;
- Team Cymru (31): through its IP to AS mapping service, it offers information drawn from a large number of BGP feeds and making it easily accessible to be queried over the Internet.

Moreover, a lot of continuous efforts are actually being performed to explore the possibility of interconnecting TopHat with other different infrastructures.

## B.3 Background measurements

TopHat regularly conducts a set of background measurements, compiling a general use archive. In this way it can provide data to those interested in the long term evolution of network topology, or to those who want to look back to a specific point in time. A user may specify a time frame when requesting a measurement, indicating the period over which he considers the measurement to be

valid: if TopHat has an archived measurement that corresponds to the requested time frame, it can serve the user with that measurement, avoiding the need to launch additional probe traffic.

In addition, the user can specify a time interval for measurement aggregation - for instance, a user might be interested in collecting a set of traceroutes between a source and a destination. Similarly, he can request summary information from aggregated data, such as average, variance, or the most frequently seen values.

Our module belong to this category of measurements. In particular, it try to estimate the target queuing delay and send the results to a specific server for further consulting. Later. a simple aggregation script can be applied to aggregate each agent result into a single global result.

# C The fastPing tool

As we stated before, the methodology is based on ICMP requests, a network-layer protocol directly encapsulated in IP. We develop a simple architecture, named fastPing, built on TopHat. We divide measurement periods (of 5 minutes by default), at the beginning of which each measurement server ask for instructions (essentially, a list of destinations/subnets and the sampling frequency, 1 Hz by default). During the measurement period, hosts are probed, in parallel, by each server. Upon reception of any new samples are for an host, beside the usual minimum, maximum, average and standard deviation, we also compute per-host percentiles of the queuing delay distribution. At the end of each measurement period, per-hosts statistics (and possibly the raw log if debug is enabled) are collected through the TopHat facility for further post-processing.

## C.1 Operational overview

In order to avoid any ambiguity and ease the following discussion, we decided to spend some time defining a vocabulary of all the terms and variables we'll use in the description of the developed module (tab. 1).

Fig. 2 shows a PlanetLab node high-level schematization. The module is installed on each agent $P_i$ and simply sends ICMP echo requests (type 8) at frequency $f$ towards an agent specific set of targets $T_{i,j}$. Measures are performed over an interval $\Delta M$ (measurement interval). At the interval timeout, the gathered results are converted into a block of files, later transferred to a specific repository server: the time spent for this operation is indicated with $\delta$ (saving interval). So we can say that the total working time of the tool can be divided into slots $S_k$ with size:

$$\Delta S = \Delta M + \delta \qquad (8)$$

Each interval $\Delta M$ is divided in cycles $C_{k,l}$ ($k$ identifies the current slot) where the tool sends a single request is sent towards every target. Each received packet is processed by a separated listener module, which extrapolates the information of interest and saves it into a local data structure before interval timeout. We anticipate that, although the tool is able to send packets only during $\Delta M$ interval in $S_k$ (before vertical dashed line in fig. 2), responses can be processed even later in $S_{k+1}$ slot.

It's clear that one problem of using PlanetLab as deployment platform lies in its testbed nature: multiple different users share the network nodes, each one using a specific virtualization of the machine sending requests with our tool. We'll propose some possible solutions to this issue.

Summarizing, the entire developed tool behaviour can be described with the Finite State Machine in fig. 3, composed by the following states:

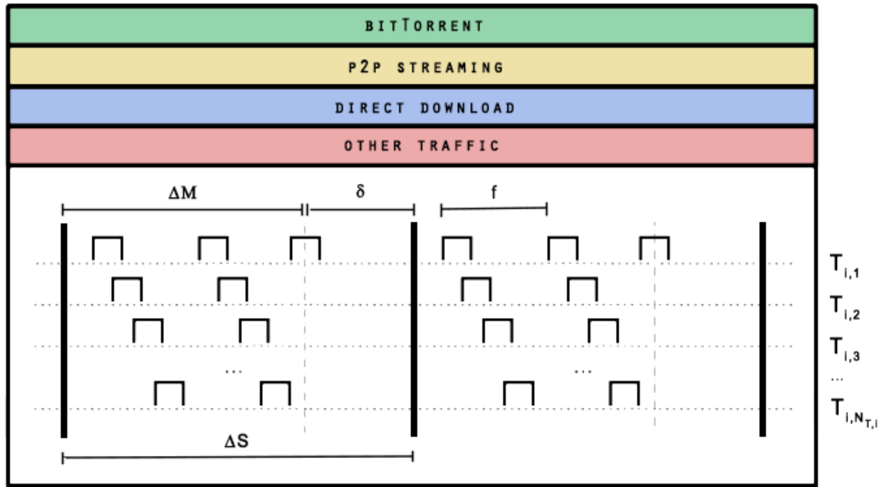


Figure 2: PlanetLab *Pi* agent

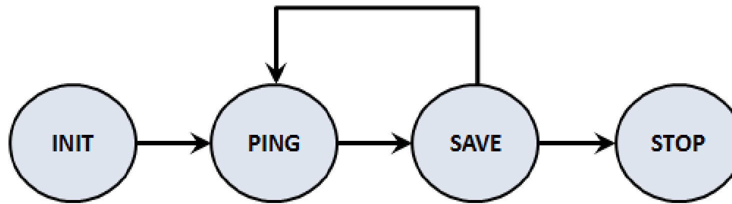


Figure 3: Pinger FSM

- **INIT:** launch setup, such as socket opening, target list loading, instantiation of local data structures and log file creation; when ready - after a startup time $\Delta init$ (not shown in fig. 2) - the tool goes into PING state to start slot $S_1$ probing;
- **PING:** measurement phase in slot $S_k$; the tool sends ICMP requests until $\Delta M$ timeout and processes responses, saving inferred statistics into local data structures and eventually inhibiting invalid targets; at timeout, it goes into SAVE state;
- **SAVE:** saving phase in slot $S_k$; the tool spends a $\delta$ interval to convert local data structures into aggregated files and send them to a remote repository; if $k \geq N_s$, it returns into PING state and starts slot $S_{k+1}$; otherwise, it goes into STOP state;
- **IDLE:** measurement loop termination; on-demand measurements should still be allowed.

In other words, the tool, after a short initialization interval, simply swings between PING and SAVE states, until a number of $N_s$ probing slots have been successfully completed.

### C.1.1 Measurement interval size

The first question to pose is about the$\Delta S$ size: we want to find its right dimension according to (2.7). The $\delta$ interval depends from target number for a specific agent and it can be changed only

reducing $N_{T,i}$ for $P_i$ agent. Its time complexity $O(N_{T,i})$ is due to the conversion phase which forces the analysis of a local data structure (containing the actual statistics) positively related to target number. Something similar happens for $\Delta init$, where the dependency from $N_{T,i}$ is related to the target list loading operation in INIT state.

This is not true for $\Delta M$, where we have many degrees of freedom to setup. The choice of measurement interval $\Delta M$ size can be taken according on different factors. Considering the expression:

$$p = \frac{\Delta M}{T} \Rightarrow \Delta M = p \cdot T = \frac{p}{f}$$

we could obtain $\Delta M$ interval setting the number of ICMP requests toward a single target and the probing period or frequency. For example, if $f = 1Hz$ - that is $T = 1s$ - and $p = 300$, we'll have $\Delta M = 300s$.

Another approach is based on considering the total number of ICMP requests ($p_{TOT}$) we'd like to send during the measurement slot $\Delta M$. Certainly we can also set this interval and calculate other parameters such as the probing frequency. Considering the following expression, for instance:

$$p_{TOT} = p \cdot N_{T,i} \Rightarrow f = \frac{p_{TOT}}{\Delta M} = \frac{p}{\Delta M} \cdot N_{T,i}$$

with $\Delta M = 300s$, $p = 100$ and $N_{T,i} = 3$, we'll still obtain $f = 1Hz$. In addition, we could consider the total number of requests sent during all the $N_S$ measurement slots.

To allow coherent aggregation of the obtained results, it is necessary that all the agents perform the measurements with the same settings. This is why $\Delta M$, $p$, $p_{TOT}$, $f$ and $T$ do not depend on $S_k$ slot or $P_i$ agent: once set, they're the same for all the nodes in the system. On the other hand, this is not true for $N_{T,i}$: setting up conveniently the tool target list, we can specify a different set or target for each agent.

## C.2 Tool architecture

As we outlined before, the tool was designed with two clear principles in mind:

- **probing speed:** maximizing the measurement throughput (number of measurements per second) in a best-effort approach;
- **probing breadth:** maximizing the cardinality of the target set for an agent $P_i$.

In fig. 4 is shown the overall system architecture. The tool exploits TopHat deployment and uploading capabilities to respectively distribute and execute the measurement agents over PlanetLab and also upload the statistics in the central repository at the end of each slot $S_k$. Specifically, fastPing is logically composed by multiple Python modules:

- **timing module:** core module of the tool; according to fig. 3 FSM, it supervises the system operations, performing modules synchronization and splitting the full measurement interval into slots;
- **probing module:** sends ICMP requests in a continuous way towards an assigned target set;
- **listener module:** analyses ICMP reply and statistics inference to the statistics module;
- **statistics module:** infers the statistics of interest at reply reception or at the end of a measurement slot, also converting the tool local data structures into uploadable files;
- **inhibition module:** enables or disables the probing towards a certain target according to received (or not received) replies;

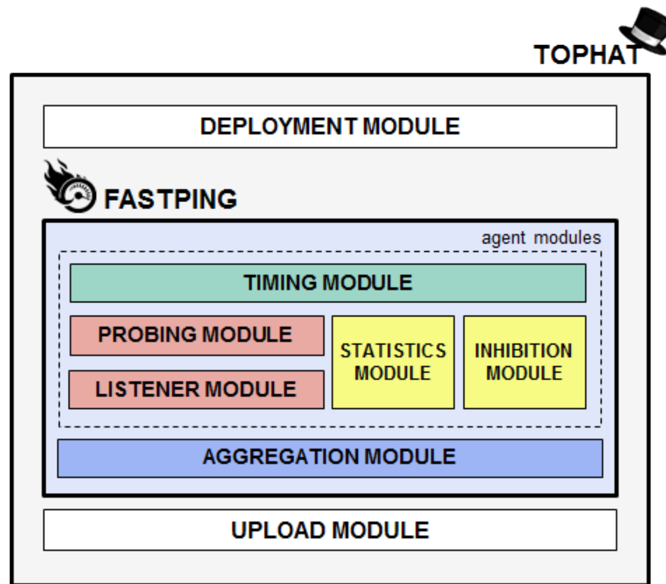


Figure 4: fastPing architecture

- **aggregation module:** aggregates the single nodes results into a single output at the end of the full measurement interval.

While the first five modules are deployed on each PlanetLab node - thus, we could refer to them as “agent modules” - the “aggregation module” is unique for the entire system and it’s executed on the repository node when all the nodes terminated their tasks.

According to the “probing speed” principle, since it’s not possible to parallelize the ICMP sending and receiving operations, we decided to structure the probing module as a single thread: using a specific thread for each target (or target group) would have indeed added useless overhead and annoying synchronization issues.

A multi-thread solution also collides with the “probing breadth” principle: increasing the number of threads - that is the number of targets - the tool would sooner or later reach limit imposed by the operating system. This limit - depending from lots of parameters, such as the “/proc/sys/kernel/threads-max” field in linux environments - is remarkably lower than the scale we intend to reach with this work (sec. **??**).

Trying to describe the complete TopHat architecture goes beyond our purposes. Thus, in the following we limit ourselves to an high-level description of the developed modules, referring to sec. E for further technical details.

### C.2.1 The timing module

The module manages the other modules tasks according to the Moore state machine described in fig. 3, where output values - in this case, the sequence of performed operations - are determined solely by the current state.

At fastPing startup (INIT state), it pre-emptively loads the target list associated to the specific agent (PlanetLab node), assigns a numeric ID to each target and waits a random time interval before starting the first measurement: in this way we avoided concurrent uploads into the repository from different agents at the end of each slot $S_k$.

During fastPing normal execution (PING state), it commissions targets monitoring to the "probing module" and waits for $\Delta M$ timeout. Then, it stops probing (SAVE state) and calls the "statistics module" to convert the gathered statistics into a bunch of files to upload on the central repository through the "upload module".

At the end of the full measurement interval (STOP state) - that is after $N_S$ slots - it invokes the "aggregation module" to obtain a single system output.

See sec. E.1 for further details.

### C.2.2 The probing module

It simply sends ICMP echo requests for a $\Delta M$ interval at a specified frequency $f$. The most tricky issue is the performance variability due to the machine load on the PlanetLab nodes where the tool is executed. With the restriction of sending a same number of packets toward all the targets in order to obtain coherent measurements, we decided to use a best-effort approach. The tool tries to probe at a steady frequency $f$: if it manages to send a packet towards every destination in the target list during a period $T$, then it waits for the end of the current period (that is the start instant of the next one) to avoid higher frequency probing; otherwise, it instantly starts the following measurement. In other words, the specified $f$ does not represent an effective value, but rather an upper limit.

To avoid our probing is confused with a DoS (Denial of Service) attack, we should set $f$ to a proper value through configuration parameters - let's say $f = 1Hz$ ($T = 1s$). If a target do not answer for a certain interval, the module proceed with a temporary inhibition until next measurement cycle through "inhibition module" (sec. C.2.5).

### C.2.3 The listener module

This module simply listens to ICMP replies and infers delay statistics through the "statistics module". More precisely, when a packet is received, the thread verifies if it's valid and coming from a destination in the target list: if so, it updates on-the-fly aggregated statistics, without saving any information about the single reply. This last requirement is needed because otherwise the tool would saturate the PlanetLab node memory in no time (sec. C.5.6).

Furthermore, if an irregular reply is processed, the associated target is temporary or permanently inhibited, according to the received packet code (sec. C.2.5).

We must consider that requests sent during a slot $S_k$ could be replied during slots $S_j$ with $j > k$, especially next to the end of measurement interval $\Delta M_k$ ($\Delta M$ in slot $S_k$). If we definitively save statistics at the end of $S_k$, we are completely ignoring useful information which could be received later. Our solution consists in postponing the statistics uploading for an additional slot, simply uploading $S_k$ statistics only at the end of slot $S_{k+1}$ - that is at the end of the following measurement cycle. In this way all the packets received in a $\delta + \Delta M = \Delta S$ interval after $\Delta M_k$ termination are still considered valid. All the following received packets are ignored, because they're likely due to transmission errors.

See sec. E.2 for further details.

### C.2.4 The statistics module

The module memorizes the temporary statistics during $\Delta M$ interval, offering to the other modules an interface to update them on-the-fly and convert them into a bunch of files to upload on the server at the end of a measurement slot.

### C.2.5 The inhibition module

In case error packets are received, their feedback is used to dynamically assign the targets to one of the following lists:

- **whitelist:** hosts that are continuously probed during the current slot;
- **greylist:** unactive hosts that will be probed again in the next slot;
- **blacklist:** administratively prohibited hosts that will not be probed again.

As we stated in sec. C.5.2, the "state" field identifies the target state at the end of $C_{k,l}$ cycle, that is the list associated to that particular target. If a target does not answer after a fixed timeout, it is greylisted ($state = 0.1$). The tool behaviour depends from "type" and "code" fields in the reply header (fig. 5):

- **type 0:** valid ICMP echo reply; the answer timeout is reset;
- **type 3:** destination unreachable; if $code = 9$ (network administratively prohibited), $code = 10$ (host administratively prohibited) or $code = 13$ (communication administratively prohibited) the target is blacklisted; otherwise it is greylisted;
- **other types:** invalid ICMP echo reply; the answer timeout is not reset.

Notice that a type-3 reply includes in the payload the IP header and first 8 bytes of original datagram's data. Through this information, it's easily possible to identify the target we want to inhibit.

### C.2.6 The deployment module

This module is part of TopHat architecture and is responsible of the agents deployment when a PlanetLab node is added to a slice. First of all, it downloads all the dependencies needed by the tool to work correctly and then, when all the preconditions are satisfied, it automatically executes it, starting the required measurements.

### C.2.7 The uploader module

The module uploads into the system repository the output files containing aggregated statistics. All uploading requests are stored into a local memory before being served. The other modules simply commission a transmission operation to the "uploader" and then return to their usual tasks without waiting for an answer. The "uploader" will perform the commissioned operation as soon as possible, retrying it in case of accidents.

In particular, if a transmission error occurs - that is the uploading of at least one file was not successful - the module introduces a certain delay before next attempt; if another error occurs, it increments this delay exponentially and tries again; otherwise it resets the delay and waits for other assignments.

Finally, even if we use the FTP protocol for transmission, the tool modularity allows to easily change it with a more convenient one such as HTTP.

See sec. E.3 for further details.

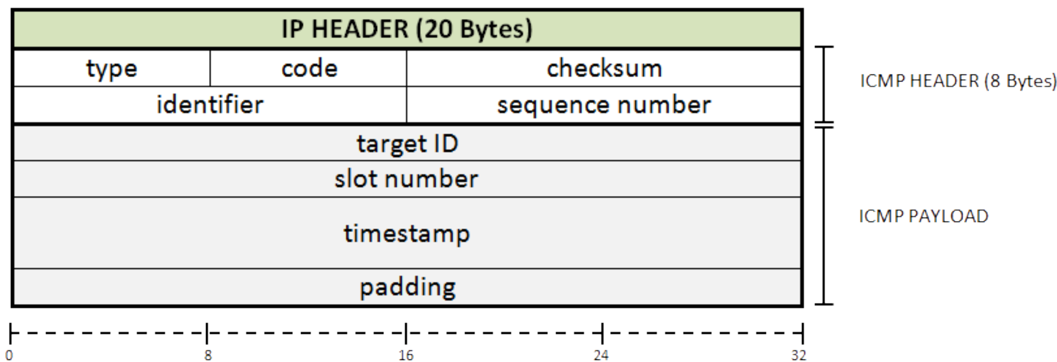


Figure 5: ICMP request/reply

## C.3 The ICMP request

According to (7), we use ICMP packets to evaluate the target queuing delay. Internet Control Message Protocol (ICMP) is one of the main protocols of the Internet Protocol Suite - as defined in RFC 792 (32) - and is typically used for diagnostic or control purposes. All ICMP packets have an 8-byte header and variable-sized data section. The first 4 bytes of the header have fixed format, while the last 4 bytes depend on the type/code of that ICMP packets. Fig. 5 shows the header and payload structure for ICMP echo request and reply: although they share the same structure, they use different values for *type* and *code* fields. We inserted inside the payload some additional information needed by the tool to work correctly: the protocol forces the target to answer with an ICMP reply with the same payload, making this information available to the sender at reception.

### HEADER

- **type (8 bits):** we use it to discriminate requests (*type=8*) from valid (*type=0*) and invalid replies;
- **code (8 bits):** specifies an ICMP subtype, giving additional information about the packet; for valid requests/replies we always have *code=0*;
- **checksum (16 bits):** used for error checking, it's calculated from the ICMP header and data, with value 0 substituted for this field;
- **identifier (16 bits):** can be used by a client to determine which echo requests are associated with the echo replies; it's randomly generated in our tool;
- **sequence number (16 bits):** same as identifier; it's incremented by one for each new request toward a certain target.

### PAYLOAD

- **target ID (32 bits):** identifier of the request destination (target);
- **slot number (32 bits):** identifies the current slot to filter no-more-useful replies;
- **timestamp (64 bits):** the request sending instant;

- **padding:** unused space for further extensions.

The 32 bit *target ID* in the payload allows us to extend the maximum probing breadth to $2^{31} - 1 = 2147483647 = O(10^9)$ targets; using the header *identifier* field (16 bit), we'd be able to probe only $2^{16} - 1 = 65536 = O(10^4)$ targets at limit, far lower than the scale aimed by the tool.

Furthermore, sending *timestamp* inside the payload is the only possible solution to allow statistics gathering, because otherwise the tool should memorise a different timestamp for each sent packet, an unfeasible operation even in presence of few targets. We pre-emptively chose a 64 bit representation, hoping that our tool will still be used after the year 2038, that is the time when a 32-bit representation overflows. In year 292,277,026,596 the Sun would be theoretically expanded to a red giant and have swallowed the Earth. Thus, problem solved!

The packets are sent through raw socket, an Internet socket that allows direct sending and receiving of packets without any protocol-specific transport layer formatting.

## C.4 The target list

The target list specifies the network entities we intend to probe. The tool accepts two different types of target:

- **single IP address:** expressed in dotted-quad notation (four integers separated by dots);
- **IP subnet:** expressed in CIDR (Classless Inter-Domain Routing) notation, a compact representation constructed from the IP address and the prefix size (a decimal number indicating the leading 1 bits in the routing prefix mask), separated by the slash ('/') character.

For each entry we need to specify the responsible PlanetLab node hostname. In this way even if the list is the same for all nodes, each one of them will only load its associated targets. Notice that target loading is performed only once: to change the entities associated to a certain node a full reset of the tool is required. Having a single list simplifies a lot the deployment operations, because the tool package is the same for all the agents in the system.

During loading operations, a numeric identifier is associated to each target ("*target ID*" field in fig. 5): we use it to find and eventually update its statistics without scanning all the target list. It's also possible to only consider a part of the full list specifying the first and last ID of interest in the configuration parameters.

## C.5 Getting online statistics

At the end of each $S_k$ slot, the tool returns three groups of statistics:

- **high-level statistics:** overall information about agent, targets and sent/received packets behaviour;
- **low-level statistics:** detailed information for each target assigned to the agent;
- **raw statistics:** optional output, it gathers information about all received packets.

These outputs are saved into files generated while the tool is in SAVE state and later uploaded into the repository. The high-level statistics give redundant and easy to read information - some of which obtained from low-level ones - used to simplify results analysis.

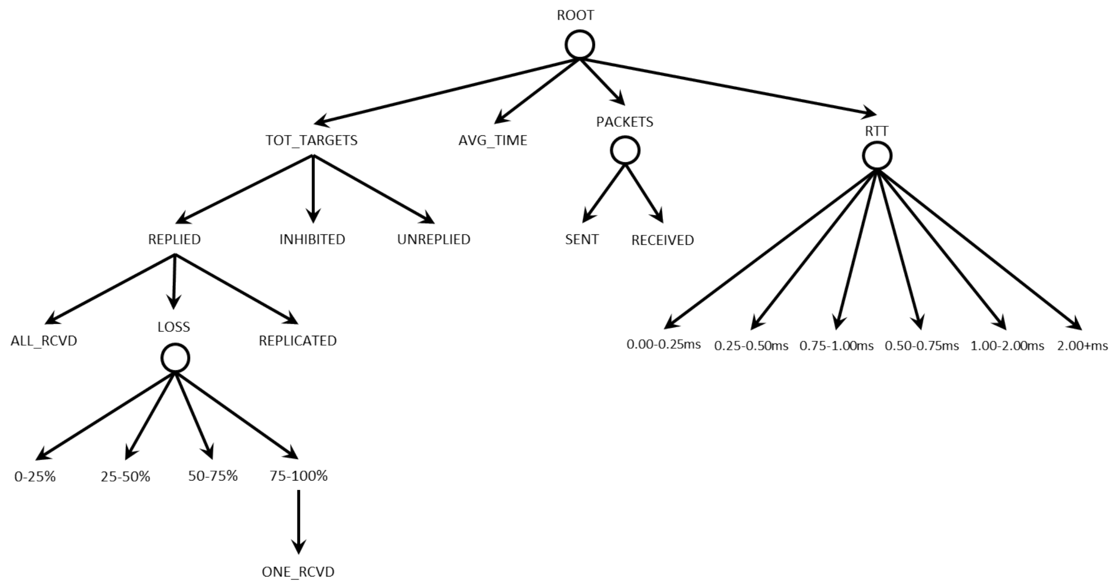


Figure 6: Global output graph

### C.5.1 High-level statistics

This group fields are organised in a tree structure, as shown in fig. 6: the more we climb down the tree, the more the information is detailed. Each tree node identifies one of the fields returned by the tool; white circles identifies fake nodes: used to support graphic representation, they simply gather two or more fields, but do not correspond to any information truly obtained from the tool. Specifically, we defined the following high-level fields:

- **avg_time:** average time the agent takes to complete a single $C_{k,l}$ cycle, that is to send one ping towards every target in the list;
- **packets:** total number of sent and received packets;
- **tot_targets:** total number of targets being associated to a specific agent;
  - **replied:** number of targets which replied at least once, divided between the ones which answered to all the ICMP requests ("**all**" field), to a certain percentage of requests ("**loss**" field) or generated inappropriate connection-level retransmissions ("**replicated**" field);
  - **unreplied:** targets which never replied;
  - **inhibited:** targets which the tool ceased to probe because they didn't reply to a specified number of consecutive requests or because of an invalid reply (sec. C.2.5).
- **RTT:** coarse-grained distribution of packets round-trip-time.

If possible, the tool also returns for some fields a percentage value (e.g., the percentage of "replied" targets among "tot_targets" number, or also "received" packets among "sent" ones).

### C.5.2 Low-level statistics

They give detailed information about the agent target set. Specifically, the following group of fields is assigned to each target ("QD" indicates queuing delay):

- **target identifiers:** numeric ID and associated IP address;
- **packets number:** number of sent and received packets;
- **QD and RTT mean:** the estimated queuing delay and round-trip-time mean;
- **QD variance:** we used the Knuth algorithm for its estimate (sec. C.6);
- **QD min and max:** the minimum and maximum detected queuing delay; we also refers to the minimum as the baseline in (2.6) expression;
- **percentiles:** we estimate 10th, 25th, 50th, 75th and 90th percentiles through P-Square algorithm (sec. C.7);
- **type:** if known, it indicates the access link type (e.g., cable, fiber, dsl); it must be statically specified in the target list;
- **state:** the target state at the end of $C_{k,l}$ cycle; with reference to fig. 5, it's generally expressed in "type.code" format (see sec. C.2.5).

### C.5.3 Queuing delay distribution

At the end of each cycle, the tool also returns the queuing delay PDF, CDF and iCDF (inverse CDF). The queuing delay can be expressed as a random variable $QD$, with values $qd \in \Omega$ and $\Omega$ continuous set. Thus, an operation of quantization is needed to discretize the interval of interest. Specifically, we indicate the interval upper limit with $QD_{max}$ , the lower limit with $QD_{min}$ and its size with $\Delta q$: the $[QD_{min}, QD_{max}]$ interval is then divided into subintervals $q_i$ with lower bound $q_{l,i}$ and upper bound $q_{u,i}$. Each queuing delay value is approximated to the nearest $q_{u,i}$. All queuing delays lower than $QD_{min}$ (upper than $QD_{max}$) are approximated to $QD_{min}$ ($QD_{max}$). In particular, the iCDF -$P(QD > qd)$, that is the probability a queuing delay exceed a certain value - allows an intuitive graphic representation of the bufferbloat issue.

### C.5.4 Raw statistics

If specified in the configuration file, the tool also saves for each received packet:

- **target ID:** the ID of the ICMP reply sender; same of fig. 5;
- **target IP:** the IP address of the ICMP reply sender;
- **sequence number:** same of fig. 5;
- **slot number:** same of fig. 5;
- **eRTT:** estimated round-trip-time for the ICMP request/reply pair.

Not only the operation of saving this information is optional, but it's absolutely not advised, because of the excessive memory space it requires. For this reason, it should be used only for debugging and inhibited through configuration file during the execution of the tool in a real environment. In every case, because it's not possible to forecast the magnitude of these information, it is directly written in a separated file on the agent local memory and not copied into the repository.

### C.5.5 Results aggregation

The system distributed nature implies the need of an aggregation operation at the end of the foreseen $N_S$ measurement slots. As we already know, the outputs are saved into a repository at the end of each slot $S_k$. Thus, when all the agents complete their execution, all the results will be gathered on a single node of the system. A specific aggregation script has been designed to join these outputs into a single bunch of files, representative of each node targets (that is the complete target list). Specifically, one queuing delays distribution, low-level and high-level statistics file are created for the entire system.

### C.5.6 Memory occupation

The tool works on local data structures which are converted into files at the end of each cycle (exclusive of raw statistics). In this way, the probing speed is increased significantly, avoiding intensive and time-consuming file operations. Even if the data structures memory occupation is remarkable, it stays constant during all the tool lifecycle, thus avoiding possible aging issues.

This is not true for files: a new bunch of files with fixed size is generated at the end of each slot $S_k$, increasing the total memory occupation in a linear way, but always of a fixed quantity. With reference to fig. 1, for a single slot $S_k$, the low-level statistics memory occupation - depending from the number of targets - is $O(N_{T,i})$, while it's $O(1)$ for high-level statistics and queuing delay distribution files and $O(p_{TOT})$ for raw statistics. Thus, during the complete tool lifecycle, the total memory occupation is respectively $O(N_{T,i} \cdot N_S)$,$O(N_S)$ and $O(p_{TOT} \cdot N_S)$.

In a real execution, the size of a queuing delay distribution file is almost *4 KB* for each slot; instead the size of an high-level statistics file is almost *12 KB*. Knowing that each target requires almost *100 B* in low-level statistics file and each packet almost *20 B* in raw statistics file, we can make a comparison between these two files: set $N_T = 10000$, assumed that all the targets always replies, after only 5 cycles - that is *5 s* with $f = 1Hz$ - they reach the same size (*1 MB*). Considering that in default settings $\Delta M = 300s$, the raw statistics file reaches a *60 MB* memory occupation in only one slot (with an increment of almost *6000%*).

## C.6 Knuth incremental algorithm (online variance)

Algorithms for calculating variance play an important role in statistical computing. It is often useful to be able to compute the variance in a single pass, inspecting each value $x_i$ only once; for example, when the data are being collected without enough storage to keep all the values (our case), or when costs of memory access dominate those of computation. For such an online algorithm, a recurrence relation is required between quantities from which the required statistics can be calculated in a numerically stable fashion.

The following formulas can be used to update the mean and (estimated) variance of the sequence, for an additional element $x_{new}$. Here, $\bar{x}_n$ denotes the sample mean of the first $n$ samples $(x_1, ..., x_n)$ and $s_n^2$ their sample variance:

$$\bar{x}_n = \frac{(n-1)\bar{x}_{n-1} + x_n}{n} = \bar{x}_{n-1} + \frac{x_n - \bar{x}_{n-1}}{n}$$

$$s_n^2 = \frac{(n-2)}{(n-1)} s_{n-1}^2 + \frac{(x_n - \bar{x}_{n-1})^2}{n}, \quad n > 1$$

It turns out that a more suitable quantity for updating is the sum of squares of differences from the (current) mean, $\sum_{i=1}^{n}(x_i - \bar{x}_n)^2$, here denoted $M_{2,n}$:

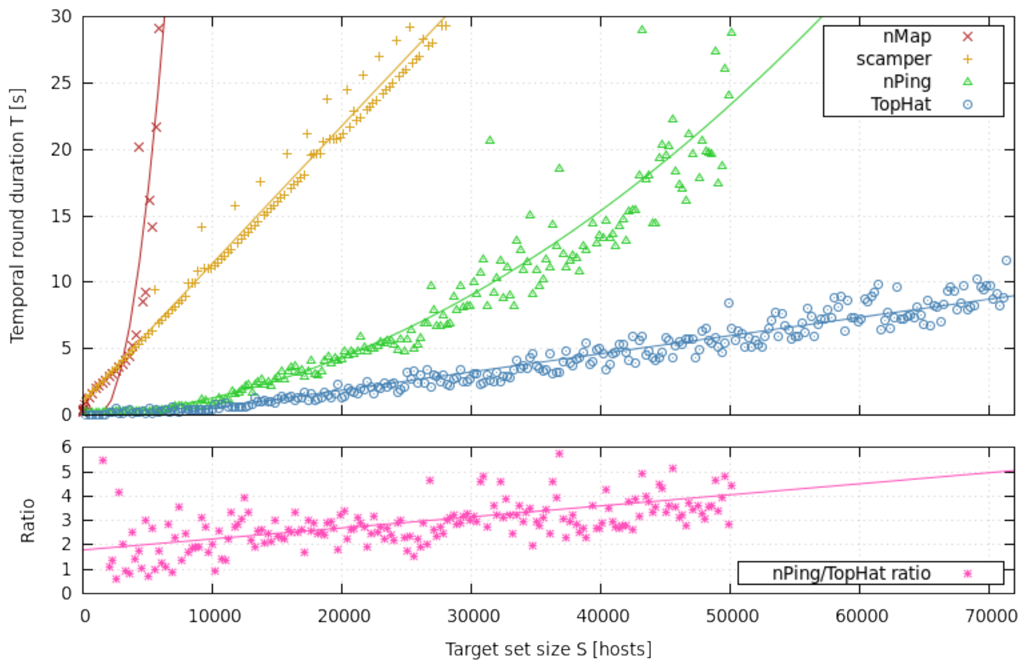


Figure 7: Bufferbloat scanner scalability

$$M_{2,n} = M_{2,n-1} + (x_n - \bar{x}_{n-1}) \cdot (x_n - \bar{x}_n) \Rightarrow s_n^2 = \frac{M_{2,n}}{n-1}$$

A numerically stable algorithm - due to Donald E. Knuth (37)- is given below.

```
def online_variance(data):
        n = 0
        mean = 0
        M2 = 0

        for x in data:
                n = n + 1
                delta = x - mean
                mean = mean + delta/n
                M2 = M2 + delta*(x - mean)

        variance = M2/(n - 1)
        return variance
```

This algorithm is much less prone to loss of precision due to massive cancellation, but might not be as efficient because of the division operation inside the loop. For a particularly robust two-pass algorithm for computing the variance, first compute and subtract an estimate of the mean, and then use this algorithm on the residuals.

## C.7 P-Square algorithm (online percentiles)

Reference: (38)

## D Scalability

To give a rough idea of the tool performance, and speculate on the achievable measurement scale, we report in fig. 7 the time needed (T, y-axis) by our tool to contact a growing number of hosts (S, x-axis). Our performance analysis focuses on the following tools:

- **nMap:** open source tool for network exploration and security auditing, originally written by Gordon Lyon (33);

- **nPing:** open source tool for large-scale ping scanning based on the Nmap Scripting Engine (34);

- **scamper:** utility providing parallel Internet measurement techniques (such as the well-known traceroute and ping techniques) and developed by Matthew Luckie (35).

According to the "probing speed" principle (sec. C.2), we tried to force a best-effort behaviour in all the compared tools. In particular, we disabled port scanning and DNS resolution for *nMap*, reducing the RTT timeout to a suggested value (100 ms) and thus sacrificing some accuracy for speed. Similarly, we imposed the highest packet-per-second rate for *scamper* and *nPing* and inhibited the output generation (quiet mode) for this last one.

Tab. 2 tabulates the spatial set size S that can be achieved in rounds having fixed temporal duration $T \in [1, 10]\,s$. We already gather that our ICMP module introduces non-negligible gains with respect to *nPing*. More precisely, the gain is lower for smaller round durations and grows with the target set size. Shown in the picture, fitting of the experimental data exhibit a linear dependency between R and S for *TopHat*. This instead does not hold for *nPing*, where for $S > 50k$ target sizes, the duration of a cycle explodes. Hence, the gains for $S > 50k$ shown in the lower plot of fig. 7 are conservative in that they are based on real data gathered for $S < 50k$. With back of the envelope calculation, we see that the core ICMP scales to about 106 probes-per-second considering a set of 100-500 OneLab/PlanetLab machines targeting 10k-2k hosts each – which fits our initial goals of high-frequency (1 pkt/s to $10^6$ hosts), or Internet-scale (1 pkt/h to $3.6 \cdot 10^9$ hosts) scans.

The reason for the better performance of our pinger module lays in the fact that the other tools, even if designed for large-scale measurements, were not developed for continuous probing and with speed requirements in mind. Our tool is not limited to immediately return an output, but it postpones this operation at the end of each slot. It also offers a better RTT timeout management, never remaining idle while waiting a reply. Finally, being the tool designed with a specific aim in mind and not for general purposes, it doesn't perform any additional operation (e.g., port scanning) which would only have the effect to increase the overall overhead. The cost to pay is the considerable (but still steady) amount of memory required to temporary and permanently memorize the statistics of interest (sec. C.5.6).

## E Tool infrastructure technical details

This section has to be intended as an extension of sec. C.2, where we added all the technical details we overlooked to not excessively overload the previous description. In this section we refer with the word "module" to all the independent entities executed in TopHat at stratup and corresponding to one or more of the modules defined previously in sec. 4.Fig. C.5.2 shows the interaction between the developed modules, that will be described in the following section.

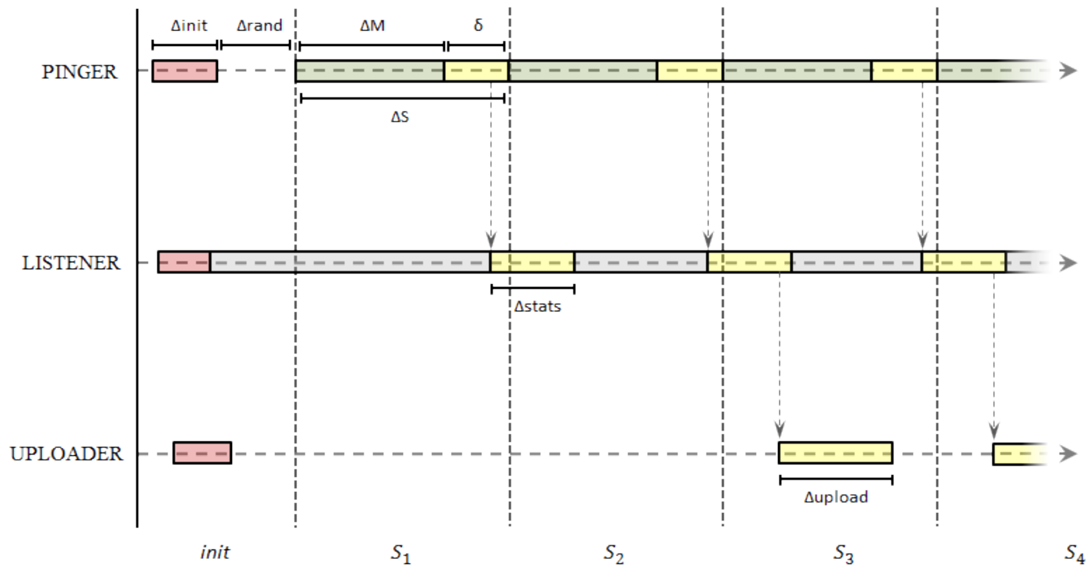


Figure 8: Low-level fastPing architecture

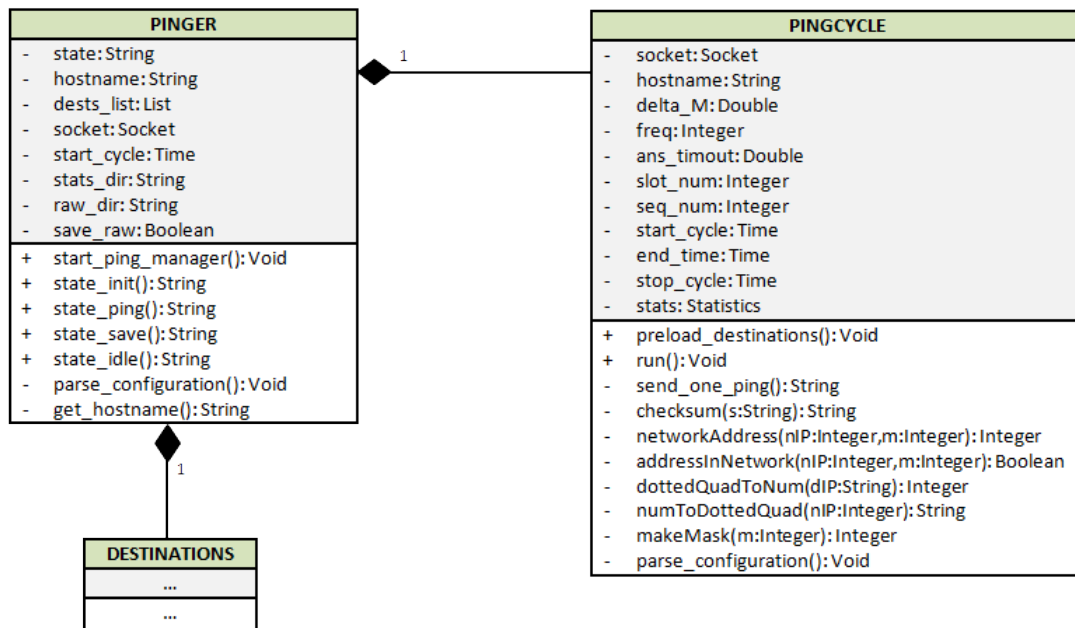


Figure 9: Pinger module class diagram

## E.1 The pinger module

As shown in fig. 9, the module is actually composed by two distinct classes, one strictly contained into the other:

- **Pinger:** it manages the tool lifecycle according to the FSM in fig. 3 (corresponding to "timing" module);
- **PingCycle:** it performs the real active measurement cycle, sending ICMP requests to the specified target set (corresponding to "probing" module)

Separating the tool lifecycle management from the probing activity allows to easily modify one without changing the other. For example, we may change the measurement methodology sending a different packet, but using the same state-machine.

The "*start_ping_manager*" method is required by TopHat to successfully load and execute the module. Each FSM state corresponds to a specific method in the *PINGER* class, which determinates the associated output in terms of sequence of operations. The *state* variable tracks down the tool current state. During the INIT state, beyond loading the target list (*"preload_destinations*" method) into a *dests_list* object (sez. XXX), the module also instanziates the data structure shared with all the other modules and used to temporary memorise the available statistics (sez. XXX).

Similarly, during PING state the "*run*" method is invoked once to perform probing until the ending of $\Delta M$ interval. As we've already stated in sez. C.2.1, the module works in a best-effort approach, trying to probe at a steady frequency *freq*: if it manages to send a request ("*send_one_ping*" method) towards every assigned destination before current $T$ interval termination (*end_time*), then it waits for the start of the next period and starts sending packets with the following sequence number (*seq_num+1*); otherwise, it doesn't wait and instantly sends the next packet. Notice that *end_time* differs from *stop_cycle*, that is the ending instant of *delta_M* and entails the $PING \rightarrow SAVE$ transition.

During SAVE state, at the end of the $S_k$ measurement cycle with $k$ identified by the *slot_num* variable, statistics are converted into files and sent to the *stats_dir* directory on the repository. If *save_raw* is set to *true*, raw traces are also saved and sent to *raw_dir* directory.

The "*PingCycle*" class also offers methods to manage IP addresses and subnetworks:

- **networkAddress(nIP,m):** it returns the IP address associated to a network with *m*-bits mask (*nNet*);
- **addressInNetwork(nIP,nNet,m):** it verifies if an address *nIP* is part of a specific network *nNet* with a known *m*-bits mask.
- **dottedQuadToNum(dIP):** it converts an IP address from its quad-dotted notation to the numeric one;
- **numToDottedQuad(nIP):** it converts an IP address from its numeric notation to the quad-dotted one;
- **makeMask(m):** it returns a network *m*-bits mask.

It's easy to notice that *nIP* - such as *nNet* - identifies an IP address expressed in a numeric notation, while *dIP* identifies an address in a quad-dotted one.

Finally, the "*parse_configuration*" and "*get_hostname*" methods allow, respectively, the conversion of configuration parameters into a proper format and the retrieval of the hostname of the machine running the agent.

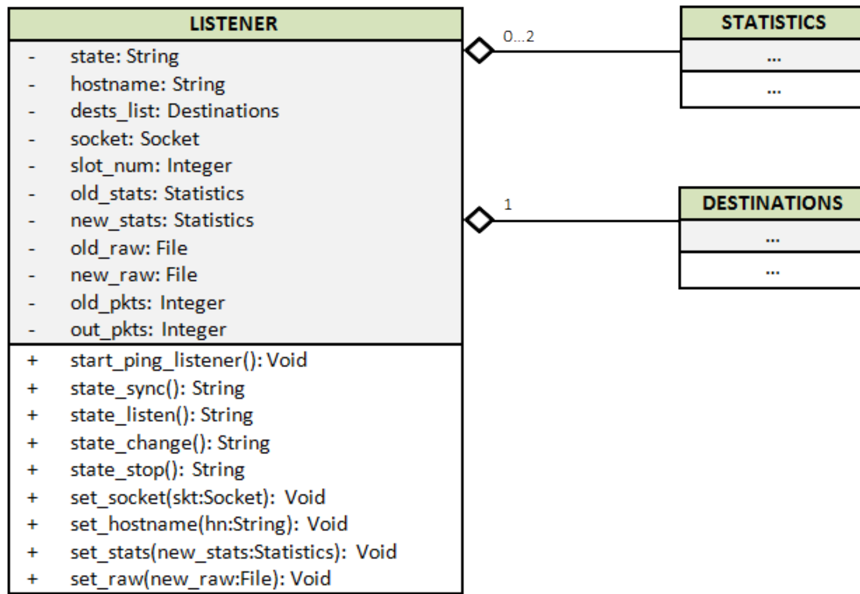


Figure 10: Listener module class diagram

## E.2 The listener module

Even in this case, TopHat executes the module invoking the "*start_ping_listener*" mehod. Its behaviour can still be described by the fig. 3 FSM, but with different operations associated to each state - this explains the employment of different names. Notice how the operations associated to each state are performed always after the PINGER tasks termination.

In SYNC state, the module waits for PINGER initialization: this last one creates the data structures shared by all the other modules (*dest_list*, *socket*, *old_stats*, *new_stats*, *old_raw*, *new_raw*) and passes the references to them through a bunch of "*set*" methods. This is highlighted by the loose inclusion in fig. 10 for "*Statistics*" and "*Destinations*" classes.

In LISTEN state, the module waits for ICMP replies and updates statistics - through "*Statistics*" class methods - when a new packet is received.

In CHANGE state, the module performs the following operations:

1. it converts *old_stats* object into a bunch of files and commissions their uploading into the repository;

2. it sets *old_stats*=*new_stats* to allow during the following cycle;

3. it sets *new_stats* to the new statistics object created by the PINGER module to start information gathering for the current cycle.

In other words, we used two data structures to memorise the halfway statistics: the first one for $S_{k-1}$ statistics; the other for $S_k$ ones. At the end of $\Delta M_k$, only $S_{k-1}$ statistics are processed, while $S_k$ ones can be updated for an other $\Delta S$ interval. Same considerations are still valid for the raw files. This also explains the meaning od *old_pkts* and *out_pkts* variables: both used for debugging, they respectively refers to the number of $S_{k-1}$ packets and $S_j$ packets - with $j < k-1$ and therefore invalid - received during slot $S_k$.

### E.3 The UPLOADER module

TopHat executes the module invoking the "*start_ping*_uploader" method. During INIT state, the module checks if the local output directory really exists ("*check_directory*" method): if not, the directory is created. Each time a node intends to upload into the repository the slot $S_k$ statistics, it first invokes the "*conv_stats*" method to convert them into a bunch of files and then the "*upload_file*" method to insert these files into the "*memory*" object.

In PROCESS state, the module tries to fulfil commissioned uploading, removing the file from the list in case of success. As we've already stated, in case of error - the last detected one identified by the "*last_error*" variable, used as flag - the tool introduce to actual "*timestamp*" a delay ("*backoff*") increased exponentially at every retry.

The "*FTPupload*" class simply implements the file transfer using the FTP protocol with the parameters specified in the tool configuration file. The "*upload*" method tries to upload all the files in memory and returns a list of all the contingent errors.

## References

[1] http://www.sdsc.edu/News%20Items/PR022008_moma.html.

[2] http://www.caida.org/projects/ark/.

[3] http://www.netdimes.org/new/.

[4] http://internetcensus2012.bitbucket.org/.

[5] M. Allman. Comments on buefferbloat. *SIGCOMM Comput. Commun. Rev.*, 43(1), Jan 2012.

[6] S. Bauer, D. Clark, and W. Lehr. Powerboost. In *ACM SIGCOMM Workshop on Home networks*. ACM, 2011.

[7] Z. Bischof, J. Otto, M. Sanchez, J. Rula, D. Chones, and F. Bustamante. Crowdsourcing ISP characterization to the network edge. In *ACM SIGCOMM Workshop on Measurements Up the STack (W-MUST'11)*, 2011.

[8] Z. S. Bischof, J. S. Otto, and F. E. Bustamante. Up, down and around the stack: ISP characterization from network intensive applications. In *ACM SIGCOMM Workshop on Measurements Up the STack (W-MUST'12)*, 2012.

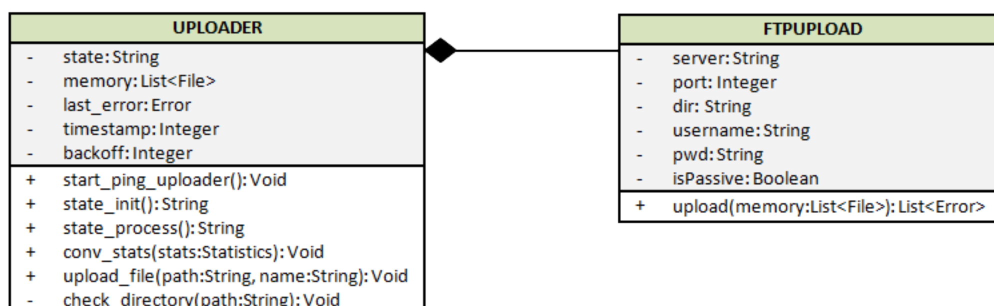


Figure 11: Uploader module class diagram

[9] T. Bourgeau, J. Auge, and T. Friedman. Tophat: supporting experiments through measurement infrastructure federation. In *TridentCom*, 2010.

[10] C. Chirichella and D. Rossi. To the moon and back: are internet buerbloat delays really that large. In *IEEE INFOCOM Workshop on Trac Measurement and Analysis (TMA)*, 2013.

[11] C. Chirichella, D. Rossi, C. Testa, T. Friedman, and A. Pescape. Remotely gauging upstream buerbloat delays. In *PAM*, 2013.

[12] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson. Fathom: a browser-based network measurement platform. In *ACM IMC*, 2012.

[13] S. Gangam, J. Chandrashekar, I. Cunha, and J. Kurose. Estimating TCP latency approximately with passive measurements. In *PAM*, 2013.

[14] J. Gettys and K. Nichols. Buerbloat: Dark buffers in the internet. *Communications of the ACM*, 55(1):57-65, 2012.

[15] H. Jiang, Y. Wang, K. Lee, and I. Rhee. Tackling buerbloat in 3G/4G networks. In *ACM IMC*, 2012.

[16] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: Illuminating the edge network. In *ACM IMC*, 2010.

[17] D. Leonard and D. Loguinov. Demystifying service discovery: implementing an internet-wide scanner. In *ACM IMC*, 2010.

[18] G. Maier, F. Schneider, and A. Feldmann. Nat usage in residential broadband networks. In *PAM*, 2011.

[19] Y. Shavitt and U. Weinsberg. Quantifying the importance of vantage points distribution in internet topology measurements. In *INFOCOM 2009*, 2009.

[20] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescape. Broadband internet performance: a view from the gateway. In *ACM SIGCOMM*, 2011.

[21] Floyd, S. and Jacobson, V. Random Early Detection gateways for congestion avoidance. In IEEE/ACM Transactions on Networking, Aug. 1993.

[22] Nagle, J. On packet switches with infinite storage. In Proceedings of Network Working Group RFC 9790, Dec. 1985.

[23] Braden, R. et al. Recommendations on queue management and congestion avoidance in the Internet. RFC2309, April 1998.

[24] Cheng Jin, David X Wei, and Steven H Low. Fast TCP: motivation, architecture, algorithms, performance. In INFOCOM, 2004. Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies, volume 4, pages 2490–2501. IEEE, 2004.

[25] Stanislav Shalunov, Greg Hazel, Janardhan Iyengar, and Mirja Kuehlewind. Low extra delay background transport (LEDBAT). draftietf-ledbat-congestion-04.txt, 2010.

[26] The OneLab experimental facility. http://www.onelab.eu/.

[27] B. Augustin, X. Cuvellier, B. Orgogozo, F. Viger, T. Friedman, M. Latapy, C. Magnien, and R. Teixeira. Avoiding traceroute anomalies with Paris traceroute. In Proc. ACM IMC, 2006.

[28] Y. Shavitt and E. Shir. DIMES: let the internet measure itself. ACM SIGCOMM Comput. Commun. Rev., 2005.

[29] SONoMA measurement infrastructure. http://www.complex.elte.hu/sonoma/.

[30] D. Morato, E. Magana, M. Izal, J. Aracil, F. Naranjo, F. Astiz, U. Alonso, I. Csabai, P. Haga, G. Simon, J. Steger, and G. Vattay. The European Traffic Observatory Measurement Infraestructure (ETOMIC): A testbed for universal active and passive measurements. In Proc. Tridentcom, 2005.

[31] Team Cymru IP to ASN mapping service. http://www.team-cymru.org/Services/ip-to-asn.html.

[32] RFC 792, Internet Control Message Protocol.

[33] Fyodor Lyon, Gordon. Nmap Network Scanning: The Official Nmap Project Guide to Network Discovery and Security Scanning. Insecure.com LLC. p. 468, January 2009.

[34] http://nmap.org/nping/

[35] M. Luckie. Scamper: a Scalable and Extensible Packet Prober for Active Measurement of the Internet. Proceedings of the 10th ACM SIGCOMM conference on Internet measurement (IMC), Melbourne, Australia, pp. 239-245, November 2010.

[36] S. Sundaresan, W. de Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescap. Broadband internet performance: a view from the gateway. In ACM SIGCOMM, 2011.

[37] Donald E. Knuth (1998). The Art of Computer Programming, volume 2: Seminumerical Algorithms, 3rd edn., p. 232. Boston: Addison-Wesley.

[38] R. Jain and I. Chlamtac, The P-Square Algorithm for Dynamic Calculation of Percentiles and Histograms without Storing Observations, Communications of the ACM, October 1985.

| | | | | |
|---|---|---|---|---|
| **$P_i$** | probe i with $i \in [1, N_P]$ | | **$N_P$** | total number of probes |
| **$T_{i,j}$** | target j for probe i with $j \in [1, N_{T,i}]$ | | **$N_{T,i}$** | number of targets for $P_i$ |
| **$S_k$** | slot k with $k \in [1, N_S]$ | | **$N_S$** | total number of slots |
| **$C_{k,l}$** | cycle l in slot $S_k$ with $l \in [1, ceil(\Delta M/T)]$ | | **p** | number of ICMP requests from $P_i$ to $T_j$ in $S_k$ |
| **ΔM** | measurement interval | | **$p_{TOT}$** | total number of sent ICMP requests in $S_k$ |
| **ΔS** | slot interval | | **δ** | saving interval |
| **f** | sending frequency | | **T** | sending period |

Table 1: Module glossary

| | T | 1s | 2s | 5s | 10s |
|---|---|---|---|---|---|
| ***nPing*** | **S** | 10k | 15k | 20k | 30k |
| ***TopHat*** | **S** | 15k | 25k | 45k | 75k |

Table 2: nPing/TopHat T comparison