FLAME: Flexible Lightweight Active Measurement Environment

Artur Ziviani, Antônio Tadeu A. Gomes,
Marcos L. Kirszenblatt, and Thiago B. Cardozo

National Laboratory for Scientific Computing (LNCC)
Av. Getúlio Vargas 333, 25651-075, Petrópolis-RJ, Brazil
{ziviani,atagomes,marcoslk,thiagoc}@lncc.br

Abstract. We propose a platform for the rapid prototyping of active measurement tools to collect network characteristics. The proposed platform provides its users with basic active measurement primitives upon which sophisticated active measurement tools can be prototyped quickly, practically, and efficiently through scripts in the Lua scripting language. We validate the platform as well as show its flexibility and accuracy through experiments on a local testbed and also on Planet-lab.

Keywords: Network measurements, Rapid prototyping.

1 Introduction

Network measurements [1,2] aim at characterizing the performance, behavior, dynamics, and structure of different kinds of networks. From an implementation viewpoint, the currently available measurement tools (see [3] for a survey) are typically coded in low-level programming languages (usually C) to avoid the impact of high-level programming language features—e.g. garbage collection and exception handling mechanisms—on the accuracy of measurement results. As a consequence, most of such tools present a potentially high development time. Besides, such tools are based on very low-level network APIs (usually BSD socket-like APIs), which hinders higher levels of code reuse across tool projects—this is easily observable in the open-source codes of several publicly available measurement tools. Further, the absence of standards in these tools with respect to the collection and storage of measurement data brings inconveniences to their integrated use in the existing measurement platforms.

In this paper, we propose a platform for the rapid prototyping of *active* measurement tools, i.e. tools based on the sending of *probes* (packets with the single purpose of performing measurements) between network nodes, thus allowing the measurement of network properties along the path linking such nodes. Our proposal, named FLAME (Flexible Lightweight Active Measurement Environment),[1,2] allows the rapid prototyping of active measurement tools even if the

[1] The source code is available at `http://martin.lncc.br/main-software-flame/`

[2] Disambiguation: We recently became aware of a set of tools also called FLAME [4] to facilitate the manipulation of flow traces, a totally different purpose than ours.

T. Magedanz et al. (Eds.): TridentCom 2010, LNICST 46, pp. 526–541, 2011.

targeted network metric depends on a *cooperative* destination node, i.e. a destination node that hosts (part of) the measurement tool.

The FLAME platform is based on the distribution of measurement agents among some network nodes. Such agents send and receive probe packets in response to commands from a central manager. These agents publish the collected measurement data in a standardized way on a central repository, simplifying the management and further analysis of such data.

The FLAME platform offers its users active measurement primitives to be executed in the agents. Users can prototype active measurement tools upon such primitives in a rapid, practical, and efficient manner. The central manager is responsible for deploying such tools and starting their execution in the agents.

Tool prototyping in the FLAME platform is based on the Lua scripting language [5]. Lua is adopted in FLAME as an *extension* language: its interpreter is embedded as a library into the measurement agents. On the one hand, the Lua interpreter gives to the scripts running in the agents access to active measurement primitives through a high-level, minimalist API [6]. On the other hand, the measurement agents and the measurement API are implemented in C, preventing significant overheads in the measurement results due to the execution of Lua scripts. We validate the platform as well as show its flexibility and accuracy through experiments on a local testbed and also on the Planet-lab platform.

The remainder of the paper is organized as follows. Section 2 briefly reviews related work. The FLAME architecture is presented in Section 3. Section 4 describes the The minimalist, high-level measurement API offered by the FLAME platform. In Section 5 we give examples of tool prototypes in the FLAME platform and present experimental results conducted using such tools to validate the platform. Finally, in Section 6 we conclude the paper, also indicating some possible future work.

2 Related Work

Related work on network measurements generally targets two different kinds of results: (i) the development or improvement of measurement tools specialized in measuring a specific subset of network environment properties—Michaut and Lepage [3] provide a survey of several existing network measurement tools; (ii) the deployment of large scale measurement platforms that employ measurement tools (usually developed independently by other projects) to analyze network metrics in a more comprehensive way. Such tools and platforms are described in the following subsections.

2.1 Tools for Active Network Measurement

Each active measurement tool typically aims at evaluating network performance according to a limited set of metrics. The most used metrics and some of the tools proposed to measure them are pointed out in Table 1, adapted from [3]. This table also indicates which tools depend on the presence of cooperative destination nodes to work correctly.

Table 1. Examples of typical active measurement metrics and tools

Tools	COP	OWD	OWV	RTT	PLR	PRO	ROT	PHC	ECP	ABW
`Iperf` [7]	✓		✓		✓					
`owping` [8]	✓	✓			✓	✓				
`pchar` [9]			✓					✓		
`pathload` [10]	✓									✓
`pathrate` [11]	✓								✓	
`ping`				✓	✓					
`QoSMet` [12]	✓	✓	✓		✓	✓				
`sprobe` [13]									✓	
`sting` [14]					✓	✓				
`traceroute`							✓			
`traceroute-paris`[15]							✓			

Legend:
COP: tools that depend on cooperative destination nodes.

OWD: one-way delay; **OWV**: OWD variation; **RTT**: round-trip time;
PLR: packet loss rate; **PRO**: packet reordering; **ROT**: route tracing;
PHC: per-hop capactity; **ECP**: end-to-end link capacity;
ABW: available bandwidth.

An important point to highlight in the active measurement tools such as those in Table 1 is the low (if none) code reuse, even for those tools developed in the same research group (e.g. `pathrate` and `pathload`) or with very similar functionalities and purposes (e.g. `traceroute` and `traceroute-paris`). Another key point to remark is that generated results are provided in an *ad hoc* format (e.g. consider the extreme case of popular tools as `ping` and `traceroute`, which provide different outputs in different operating systems), making it difficult to manage this data and eventually analyze measurement results in an integrated way. As an example, although there exists some proposals to standardize log formats [16,17], few active measurement tools (e.g. `pathrate` and `pathload`) do use these standards to present the results of their measurements.

2.2 Platforms for Active Network Measurement

NIMI [18] and ETOMIC [19] are large-scale active measurement platforms that apply restrictive, domain-based trust models among measurement managers and agents. Both platforms use third-party active measurement tools as plug-ins in the measurement agents, which hinders code reuse across tools, making it harder the implementation of innovative measurement techniques. Besides, neither NIMI nor ETOMIC provides a central result repository, making it more difficult to analyze measurement results in an integrated way.

DIMES [20] goes in a different direction by employing an approach similar in concept to projects on voluntary computing (such as SETI@home [21]), rendering a potentially high coverage of the platform. The main goal of DIMES is to make available a detailed connectivity graph of the Internet. DIMES offers a XML-based language, called PENny, for the specification of the experiments. This language is rather limited in its expressiveness, however, only offering primitives for measurement of bidirectional delay and route tracing, which constrains the flexibility of the DIMES platform.

ANEMOS [22] and Flexmon [23] focus on the provisioning of central result repositories. ANEMOS allows the definition of rule-based alarms according to the measured network performance, whereas Flexmon is able to constrain probes according to the resources available at the measurement agents. Like NIMI and ETOMIC, ANEMOS and Flexmon employ third-party active measurement tools.

ATMEN [24] and NetQuest [25] aim at reducing the overhead of active network measurements. ATMEN avoids wasted measurements by judiciously reusing measurement results. NetQuest employ inference algorithms that select data collected in a measurement experiment to maximize the amount of information gathered about the properties of certain network paths, given the set of constraints about the experiments (e.g. the maximum allowed amount of issued probes). Like most of the aforementioned platforms, ATMEN uses third-party active measurement tools. The way such tools are implemented/used in NetQuest is not presented in [25].

Scriptroute [26] is the approach we regard as closest to ours. In this platform active measurement tools are specified in Ruby, a fully object-oriented scripting language, whereas measurement agents, which interpret the Ruby scripts, are implemented in C. Like Flexmon, Scriptroute is able to constrain probes according to the resources available at the measurement agents. Nevertheless, the flexibility of Scriptroute is limited by the impossibility of implementing tools that depend on cooperative destination nodes. Furthermore, Scriptroute does not have a centralized repository in which the measurement results are stored. Thus, the gathering and output of results must be explicitly coded in the Ruby scripts that implement the tools. This approach renders scripts that tangle probing and data output functionality, hindering higher levels of reuse.

3 FLAME Architecture

The FLAME architecture is presented in Figure 1. The communication between the components of the architecture—namely the *user console*, the *measurement manager*, and the *measurement agents*—is performed through the XMPP protocol [27]. An XMPP service (comprising one or more XMPP servers, either dedicated or public ones) acts as a message bus among FLAME components. We describe the role of each FLAME component in the following.

Users issue measurement sessions to the measurement manager using a text-based console application. The measurement manager is responsible for initiating the issued measurement sessions—called *experiments* in FLAME—in the measurement agents and for storing the results obtained in these sessions in a central repository (a relational database in the current version of platform). The measurement agents are responsible for executing the experiments, caching the collected measurement results and sending them to the manager at the end of these experiments—such delay is to avoid remote repository updates to interfere with the experiments.

An experiment is specified as a script that describes the prototyped tools to be adopted in the experiment and the commands that invoke such tools. Such a

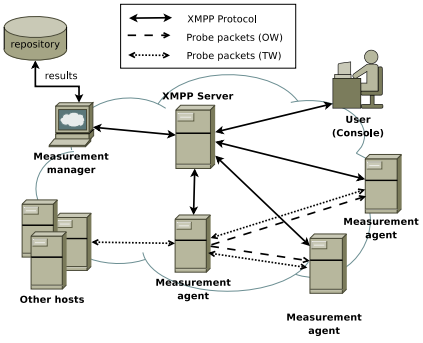


Fig. 1. FLAME architecture

script has access to the basic primitives of the measurement API implemented by the agent (see Section 4 for further details about such primitives). Both the script and the corresponding collected measurement results from an experiment are published in the central repository. This allows for better data provenance, since the prototyped tools and the commands that invoke them can be later analyzed (for instance, concerning their correctness and accuracy) based on the collected results.

It is worth noting that measurements may be performed between sets of measurement agents, or between measurement agents and ordinary hosts (i.e. hosts that do not have a measurement agent but may reply to probes, for instance a host answering ICMP requests), as depicted in Figure 1. In the case of measurements between sets of agents, besides two way (TW) measurements such as those performed between agents and ordinary hosts, it is also possible to have one-way (OW) measurements.

The use of XMPP presence controls permits the measurement manager to ask agents to perform an specific experiment in *exclusive* mode, meaning that the XMPP service will make such agents temporarily unavailable for other experiments. This functionality allows carrying out experiments that would otherwise be disturbed by concurrent experiments in the same set of agents.

4 The Minimalist, High-Level Measurement API

In the FLAME platform, experiments are specified in the Lua scripting language [5]. We chose Lua to prototype active measurement tools due to the following convenient characteristics it presents:

– Lua has a simple (procedural) syntax and a high abstraction level, thus having a great potential to reduce software development time. In the FLAME platform, the combination of Lua and a central repository allows the developer of measurement tools to focus more on the probing techniques than on the particularities of lower-level languages (e.g. memory management) and data output functionality;

– Lua presents extensible semantics (e.g. allowing the use of threads in different collaboration levels [28]) and a reduced memory footprint,[3] thus making measurement agents deployable on nodes with diverse levels of resource availability.

Each measurement agent in the platform hosts an adapted Lua interpreter that provides a sandboxing environment (like the Scriptroute platform) for the controlled execution of Lua scripts. The agents make the measurement primitives available to the Lua scripts through an API in C that is exported to the Lua interpreter with the name `lamp` (*Lua Active Measurements Primitives*).

The names of the probing operations offered by the `lamp` API follow the `send[protocol][type](...)` structure, where `protocol` indicates the protocol used for sending probes (in the current implementation, UDP, TCP, and ICMP are available), and `type` indicates if the probes are bidirectional (TW – *Two Way*), unidirectional (OW – *One Way*), or sent in a burst (PT – *Packet Train*). The TW operations do not depend on cooperative destination nodes and the results obtained with these operations are collected in the source of the experiment. In this case, the source node is responsible for sending the results to the measurement manager. The OW and PT operations depend on cooperative destination nodes. In this case, a destination node is responsible for collecting the results and sending them to the measurement manager. Such operations, however, also instruct the destination nodes to send the collected results back to the source node after the operation execution. This is important when the developer needs to implement active measurement tools that rely on successive iterations based on the feedback from the cooperative destination nodes to properly measure certain network characteristics (as in the case of `pathload`, for instance).

All probing operations of the `lamp` API return a Lua table[4] containing the collected results, in case of success. Such probing operations are extensibly parametrized, but without impacting significantly on the usability gained with a minimalist API, given that several parameters are optional and, when omitted, receive default values.

Besides probing operations, the `lamp` API also offers other operations, such as `sleep(...)` (to suspend the execution of a script for a certain amount of time), and a set of operations for querying the central repository, allowing the reuse of previously implemented tools—and collected results, like the ATMEN platform—in the context of other experiments (due to space restrictions we omitted in this paper a detailed explanation of such operations).

[3] A comparative benchmark study, available at `http://shootout.alioth.debian.org`, points out an average memory consumption 530%, 260%, and 190% larger than Lua for Ruby, Python, and Perl, respectively.

[4] Lua offers a single type of data structure, called *table*, that implements associative arrangements of the kind $\{k_1 = v_1, \ldots, k_n = v_n\}$, i.e. each value v_i stored in the structure is associated to a key k_i. Keys and values can assume any type, even though numbers and strings are the most common types of keys. Values are indexed by numeric keys in the form `table[key]` and by alphanumeric keys in the form `table[''key'']`, or simply `table.key`.

5 Experimental Validation

To validate the FLAME platform as well as to illustrate its flexibility and accuracy, we conducted a set of measurements on two experimental scenarios: the Planet-lab platform and a local testbed. We stress that these experiments are intended to validate the FLAME platform and not to provide a performance evaluation of FLAME.

The following subsections describe these experimental scenarios and compare a set of measurements collected in such scenarios with publicly available tools and equivalent tools prototyped on the FLAME platform. In the case of the publicly available tools, the measurement results were obtained by post-processing the textual output of such tools. In the case of our prototyped tools, the measurement results were obtained by querying the central repository of FLAME directly and post-processing the returned data. It is important to emphasize here that this approach to data collection encourages the untangling of probing and data output functionality, which can be observed from the compactness and neatness of the prototype sources presented in the following subsections.

5.1 Experimental Scenarios

The Planet-lab platform was used for RTT and route tracing experiments. For such experiments, we employed six nodes spread throughout the globe, one node (S1) working as the source and the other five nodes (T1, ..., T5) as targets. Such nodes are presented in Table 2.

Table 2. Source and target nodes of the Planet-lab experiments

	IP address	Domain name
Source		
S1	200.19.159.34	`planetlab1.pop-mg.rnp.br`
Targets		
T1	130.83.166.198	`host1.planetlab.informatik.tu-darmstadt.de`
T2	128.112.139.80	`alice.cs.princeton.edu`
T3	132.65.240.100	`planet1.cs.huji.ac.il`
T4	130.216.1.22	`planetlab-1.cs.auckland.ac.nz`
T5	131.112.243.102	`node2.planet-lab.titech.ac.jp`

Our local testbed was used for one-way delay and link capacity experiments. It comprises four Linux nodes (Ubuntu distribution) connected in a row topology; the two nodes at the edges of the row working as the source and destination nodes and the other two nodes as intermediate routers. For this scenario we adopted two link configurations. In the first configuration, the three links worked at 10 Mbps (referred to in this section as the "10-10-10 topology"). In the second configuration, the link in the middle of the row topology worked at 100 Mbps (the "10-100-10 topology").

5.2 RTT Measurements

For our RTT experiments in Planet-lab, we used the `ping` tool available in node S1 and prototyped an equivalent tool (`flamePing`) using the `lamp` API, deploying it in the FLAME measurement agent running on node S1. The code in Listing 1 illustrates our `flamePing` prototype.

```
1  function flamePing(params)
2    -- Ping params (and corresponding defaults)
3    local _target = params.target or "127.0.0.1"
4    local _size = params.size or 56
5    local _interval = params.interval or 250000 -- microsecond resolution
6    local _npackets = params.npackets or 10
7    local _protocol = params.protocol or "icmp"
8    local _timeout = params.timeout or 5000000 -- microsecond resolution
9    local _ttl = params.ttl or 30
10
11   for i = 1,_npackets do
12     local _response
13
14     -- Choose probing protocol (for UDP and TCP primitives,
15     --  since port is not indicated it is randomly chosen above 1024)
16     if _protocol == "icmp" then
17       _response=lamp.sendICMPTW{ip=_target, size=_size,
18                                 timeout=_timeout, ttl=_ttl}
19     elseif _protocol == "udp" then
20       _response=lamp.sendUDPTW{ip=_target, size=_size,
21                                timeout=_timeout, ttl=_ttl}
22     elseif _protocol == "tcp" then
23       _response=lamp.sendTCPTW{ip=_target, size=_size,
24                                timeout=_timeout, ttl=_ttl}
25     else
26       print("INVALID PROTOCOL:", _protocol)
27       return
28     end
29
30     -- Check host/net unreachability
31     if _response and _response.loss == ICMP_HOST_UNREACH then
32       print("DESTINATION UNREACHABLE: ", _target)
33       return
34     end
35
36     -- Wait time between probes
37     if type(_interval) == "number" then
38       lamp.sleep(_interval)
39     elseif type(_interval) == "table" then
40       lamp.sleep(_interval.func(_interval.params))
41     end
42   end --for i
43 end
```

Listing 1. FLAME ping prototype

It is worth noting in Listing 1 that our `flamePing` prototype is pretty simple, yet provides functionality for sending UDP and TCP probes besides ICMP probes (lines 14-28), and for spacing probes according to arbitrary functions (lines 36-41). In Table 5.2 we illustrate two different sets of commands that issue the execution of `flamePing` on node S1 in Planet-lab, the first one with probes uniformly spaced (as the original `ping` tool does) and the second one with probes exponentially spaced.

Table 3. Commands to execute `flamePing` over Planet-lab nodes

Probes uniformly spaced	Probes exponentially spaced

`local targets = {`
`  "130.83.166.198",`
`  "128.112.139.80",`
`  "132.65.240.100",`
`  "130.216.1.22",`
`  "131.112.243.102"`
`}`
 	
`for k,v in ipairs(targets) do`
`  flamePing{target=v}`
`end`
	`local function expdist(lambda)`
	`  local r = 0`
	`  repeat r = math.random() until(r ~= 0)`
	`  return math.floor(-math.log(r)/lambda)`
	`end`
	
	`local targets = ... -- same as left`
	
	`for k,v in ipairs(targets) do`
	`  flamePing{`
	`  target=v,`
	`  interval={ func=expdist, params=1/10000000 }`
	`  }`
	`end`

Figure 2 illustrates some RTT measurement statistics collected in Planet-lab with `ping` and `flamePing` (both uniform and exponential distributions). Such statistics comprise, for each target in Table 2, 36 interleaved executions of each tool spaced by 10 minutes, to minimize the effect of traffic synchronization. For each tool, one execution sent out 10 probes of 56 bytes for a total of 360 samples per tool and a total of 18 hours of measurement collections. The error bars in the figure show for each tool the 05- and 95-percentiles among the 360 samples, and the data points show the corresponding averages. It is important to note how similar the `ping` and `flamePing` tools characterized the network behavior with regard to RTTs among the used Planet-lab nodes.

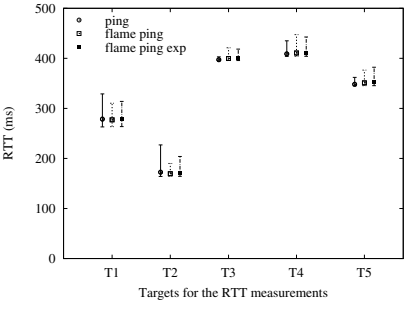


Fig. 2. RTT measurement statistics in Planet-lab

5.3 Route Tracing

For our route tracing experiments in Planet-lab, we used the `traceroute` tool available in node S1 and prototyped an equivalent tool (`flameTrace`) using the `lamp` API, deploying it in the FLAME measurement agent running on node S1. The code in Listing 2 illustrates our `flameTrace` prototype.

It is worth noting the similarity between Listing 1 and Listing 2, which becomes more apparent as the FLAME design encourages the separation between packet probing and data output/processing functionality. For triggering route

```
function flameTrace(params)
  local _target = params.target or "127.0.0.1"
  local _size = params.size or 60
  local _interval = params.interval or 250000
  local _npackets = params.npackets or 3
  local _protocol = params.protocol or "udp"
  local _timeout = params.timeout or 5
  local _maxhops = params.maxhops or 30

  for h = 1, _maxhops do
    local _response

    for i = 1,_npackets do
      -- Choose probing protocol (for UDP and TCP primitives,
      --  since port is not indicated it is randomly chosen above 1024)
      if _protocol == "icmp" then
        _response=lamp.sendICMPTW{ip=_target, size=_size,
                                  timeout=_timeout, ttl=h}
      elseif _protocol == "udp" then
        _response=lamp.sendUDPTW{ip=_target, size=_size,
                                 timeout=_timeout, ttl=h}
      elseif _protocol == "tcp" then
        _response=lamp.sendTCPTW{ip=_target, size=_size,
                                 timeout=_timeout, ttl=h}
      else
        print("INVALID PROTOCOL:", _protocol)
        return
      end

      ... -- same as lines 30-41 in flamePing
    end --for i

    -- Is current node the target?
    if (_response.remIP == _target) then break end
  end --for h
end
```

Listing 2. FLAME route tracing prototype

tracing experiments in Planet-lab using FLAME, we issued the commands below on node S1:

```
local targets = {"130.83.166.198", "128.112.139.80",
                 "132.65.240.100", "130.216.1.22", "131.112.243.102"}

for k,v in ipairs(targets) do flameTrace{target=v} end
```

Like the RTT experiments, the route tracing experiments were conducted using the `traceroute` and `flameTrace` tools in an interleaved way for a total of 18 hours. Figure 3 shows a graph representation of the paths linking the Planet-lab nodes used in the experiments, as identified by both the `traceroute` and `flameTrace` tools. The results of both tools were again quite similar, showing no changes on such routes during the experiments. The only perceived difference was on the process of route tracing between S1 and T1 nodes. In Figure 3, the dashed loop arrow on T1 indicates that in most of our experiments the `traceroute` tool couldn't realize it had arrived at the destination node, continuing to increment the TTL field of probing packets until its maximum value (30 hops). This situation didn't happen with the `flameTrace` tool in any of the experiments. We attribute this phenomenon to an unexpected situation that `traceroute` couldn't

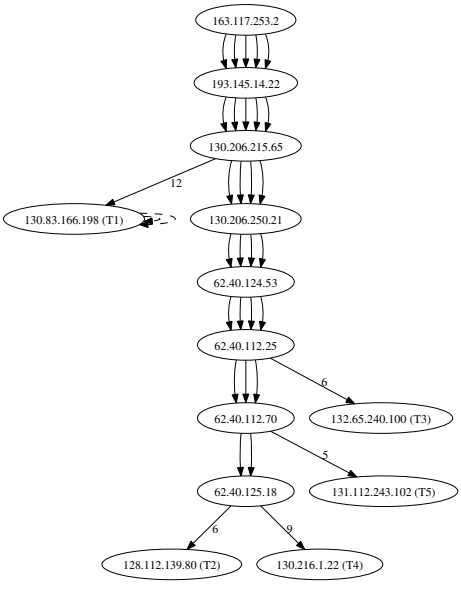


Fig. 3. Graph representation of `traceroute` and `flameTrace` results. The numbered edges indicate the number of omitted hops between the corresponding nodes.

sort out. With the rapid prototyping facilities offered by FLAME we expect such kind of situation to be easily identified and corrected by the tool developer.

5.4 One-Way Delay Measurements

Unlike the tools used for the RTT and route tracing experiments, the tools we used for one-way delay measurements depend on cooperative destination nodes. To minimize clock synchronization errors, these experiments were conducted in our local testbed with the source and destination nodes synchronizing on a local NTP server. For these experiments, we installed the `owping` tool in the source and destination nodes and prototyped an equivalent tool (`flameOWD`) using the `lamp` API, deploying it in the FLAME measurement agent running on the source node (in these experiments we also ran a management agent on the cooperative destination node). The code in Listing 3 illustrates our `flameOWD` prototype.

The single command `flameOWD{target=‘‘10.0.2.2’’, npackets=200}` was used for issuing the execution of `flameOWD` on the source node in our testbed.

Table 4 illustrates some one-way delay measurement statistics collected in our local testbed using the 10-10-10 and 10-100-10 topologies with `owping` and `flameOWD`. Such statistics comprise, for each topology, 200 probes of 56 bytes. The table shows for each tool and topology the median, 05- and 95-percentiles among the 200 samples. It is important to note that in both topologies the `flameOWD` prototype provided lower one-way delays, but the NTP estimated synchronization error (the same for both tools) rendered statistically equivalent results between the two tools.

```
1  function flameOWD(params)
2    local _target = params.target or "127.0.0.1"
3    local _size = params.size or 56
4    local _interval = params.interval or 750000
5    local _npackets = params.npackets or 5
6    local _protocol = params.protocol or "udp"
7    local _timeout = params.timeout or 5000000
8    local _port = params.port or nil
9
10   for i = 1,_npackets do
11     local _response
12
13     -- Choose probing protocol (if port is not indicated the destination node
14     -- chooses a port above 1024. In any case, for OW operations,
15     -- the source and destination nodes agree on the used port
16     -- through XMPP messages)
17     if _protocol == "udp" then
18       _response=lamp.sendUDPOW{ip=_target, size=_size,
19                                timeout=_timeout, port=_port}
20     elseif _protocol == "tcp" then
21         _response=lamp.sendTCPOW{ip=_target, size=_size,
22                                timeout=_timeout, port=_port}
23     else
24       print("INVALID PROTOCOL:", _protocol)
25       return
26     end
27
28     -- Check port unavailability and host/net unreachability
29     if _response then
30       if _response.loss == ICMP_HOST_UNREACH then
31         print("DESTINATION UNREACHABLE: ", _target)
32         return
33       elseif _response.loss == PORT_ALREADY_IN_USE then
34         print("DESTINATION PORT ALREADY IN USE: ", _target, _port)
35         return
36       end
37     end
38
39     ... -- same as lines 36-41 in flamePing
40   end --for i
41 end
```

Listing 3. FLAME one-way delay measurement prototype.

Table 4. Measured one-way delay in ms. NTP estimated synch error is ± 0.41 ms

	10-10-10 Topology			**10-100-10 Topology**		
	P05	Median	P95	P05	Median	P95
`owping`	1.28	1.39	1.50	1.10	1.21	1.32
`flameOWD`	0.87	0.91	0.98	0.46	0.51	0.58

5.5 Link Capacity Estimation

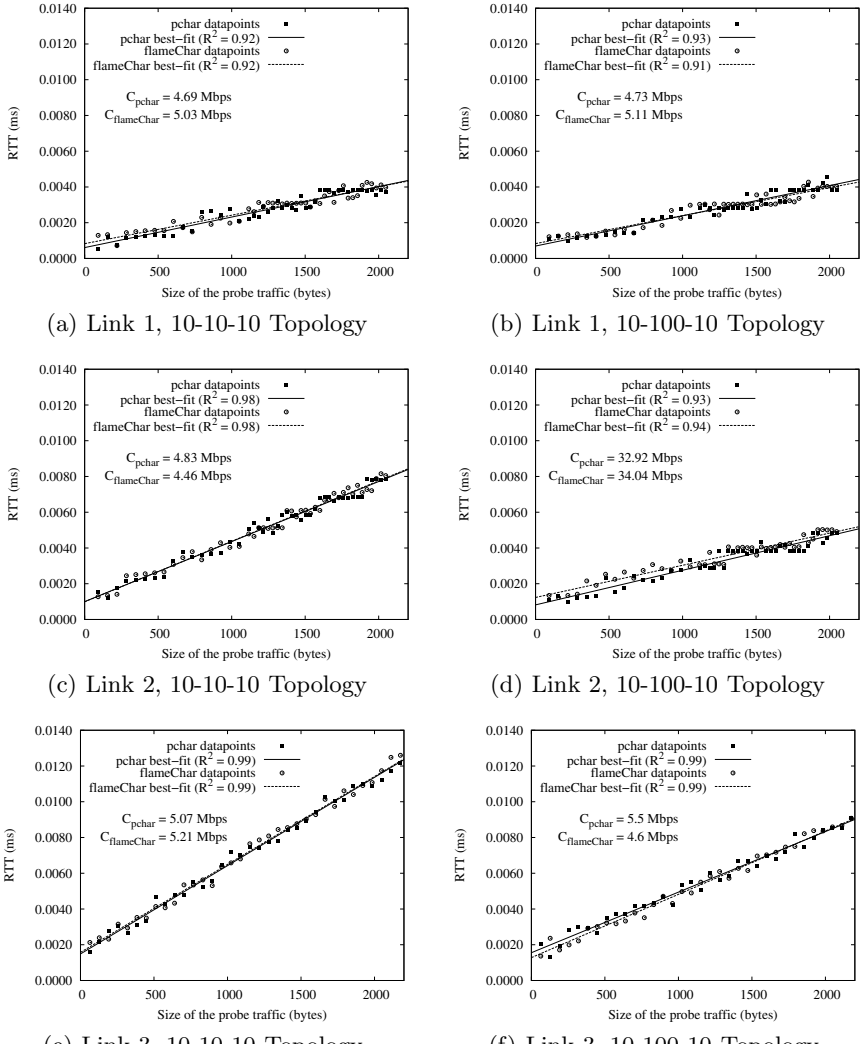
For the link capacity experiments, we employed our local testbed so that we could have more precise information about the actual link capacities we were interested in estimating. For these experiments, we installed the `pchar` tool in the source node and prototyped an equivalent tool (`flameChar`) using the `lamp` API, deploying it in the FLAME measurement agent running on the source node. The code in Listing 4 illustrates our `flameChar` prototype.

```
function flameChar(params)
  local _mtu = params.mtu or 1500 -- probe size limit
  local _increment = params.increment or 32 -- difference between probe sizes
  local _npackets = math.floor(_mtu/_increment)
  local _maxhops = params.maxhops or 30 -- maximum number of allowed hops
  local _repetitions = params.repetitions or 32 -- number of tests per hop
  local _target = params.target or "127.0.0.1"
  local _timeout = params.timeout or 3
  local _interval = param_table.interval or 500000 -- time between probes

  local _sizelist = generateProbeSizeList(_npackets, _increment)

  for h = 1, _maxhops do
    local _response
    for t = 1,_repetitions do
      for i = 1,_npackets do
        local _response

        -- Choose probing protocol (for UDP and TCP primitives,
        --  since port is not indicated it is randomly chosen above 1024)
        if _protocol == "icmp" then
          _response = lamp.sendICMPTW{ip=_target, size=_sizelist[i],
                                      ttl=h, timeout=_timeout}
        elseif _protocol == "udp" then
          _response = lamp.sendUDPTW{ip=_target, size=_sizelist[i],
                                      ttl=h, timeout=_timeout}
        elseif _protocol == "udp" then
          _response = lamp.sendTCPTW{ip=_target, size=_sizelist[i],
                                      ttl=h, timeout=_timeout}
        else
          print("INVALID PROTOCOL:", _protocol)
          return
        end

        ... -- same as lines 30-41 in flamePing
      end --for i
    end --for t

    -- Is current node the target?
    if (_response.remIP == _target) then break end
  end --for h
end

-- Function to create a shuffled package size list
function generateProbeSizeList(npackets, increment)
  ... -- implementation omitted due to space limitations
end
```

Listing 4. FLAME link capacity estimation prototype.

The single command `flameChar{target=''10.0.2.2''}` was used for issuing the execution of `flameChar` on the source node in our local testbed.

Figure 4 illustrates some link capacity measurements and related statistics collected in our local testbed using the 10-10-10 and 10-100-10 topologies with `pchar` and `flameChar`. Such measurements and statistics comprise, for each topology, a single execution of each tool. For each tool, the execution sent out 32 probes of varying sizes (from 32 to 1472 bytes in increments of 32 bytes) in a particular topology. The figure shows for each tool and topology the minimum RTTs per probe size and the "best-fit" lines obtained through the linear least squares fitting technique. As shown in [9], the slope of such lines is used for estimating the capacity of each link in both topologies. Figure 4 also shows the



(a) Link 1, 10-10-10 Topology

(b) Link 1, 10-100-10 Topology

(c) Link 2, 10-10-10 Topology

(d) Link 2, 10-100-10 Topology

(e) Link 3, 10-10-10 Topology

(f) Link 3, 10-100-10 Topology

Fig. 4. Link capacity measurements

capacity estimation per link obtained with both tools (C_{pchar} and $C_{flameChar}$). As can be seen from Figure 4, both tools returned quite similar results.

6 Conclusions

We proposed the FLAME platform for the rapid prototyping of active measurement tools and the execution of experiments using them. Overall, compared with the main characteristics of previous work, FLAME provides the following contributions: (i) an environment for the rapid prototyping of active measurement

tools based on a small but comprehensive set of probing primitives, allowing the implementation of tools that depend or not on cooperative destination nodes; (ii) a centralized repository for implicitly gathering measurement results in a common data format, encouraging the untangling of probing and data output functionality and easing the process of further analysis and comparison among different result datasets. It is also interesting to highlight the declared future goal of CAIDA's newest Ark active measurement infrastructure [29] to provide a high-level API that eases the challenges of writing measurement tools. Inline with this trend, we believe our FLAME platform achieves this goal.

As future work, we plan to implement other well-known active measurement tools to further validate the comprehensiveness of our measurement API. Moreover, we intend to evaluate the performance of the FLAME platform and its prototypes in face of other platforms such as scriptroute and low-level developed tools for active measurements. We also intend to port our FLAME platform to resource-constrained devices such as PDAs and sensors, taking profit from the low footprint of the Lua interpreter, so that we can collect measurement data on networks such as cellular, ad-hoc, disruption-tolerant, and sensor networks.

Acknowledgement

This work was supported by the Brazilian Funding Agencies FAPERJ, CNPq, CAPES, and by the Brazilian Ministry of Science and Technology (MCT).

References

1. Crovella, M., Krishnamurthy, B.: Internet Measurement: Infrastructure, Traffic and Applications. John Wiley & Sons Inc., New York (2006)
2. Ziviani, A.: Internet measurements. In: Freire, M., Pereira, M. (eds.) Encyclopedia of Internet Technologies and Applications, pp. 235–241. IGI Gobal (2007)
3. Michaut, F., Lepage, F.: Application-oriented Network Metrology: Metrics and Active Measurement Tools. IEEE Comm. Surv. & Tut. 7(2), 24 (2005)
4. Brauckhoff, D., Wagner, A., May, M.: Flame: a flow-level anomaly modeling engine. In: CSET 2008: Proceedings of the conference on Cyber security experimentation and test, pp. 1–6. USENIX Association, Berkeley (2008)
5. Ierusalimschy, R., Henrique, L., Waldemar, F., Filho, C.: Lua – an extensibleextension language. Software: Practice and Experience 26, 635–652 (1996)
6. Henning, M.: API design matters. Queue 5(4), 24–36 (2007)
7. Hsu, C., Kremer, U.: IPERF: A framework for automatic construction of performance prediction models. In: Proceedings of the Workshop on Profile and Feedback- Directed Compilation (1998)
8. Boote, J.: One–way ping, OWAMP (2009), `http://e2epi.internet2.edu/owamp`
9. Downey, A.B.: Using pathchar to estimate Internet link characteristics. In: Proceedings of the ACM SIGCOMM 1999, pp. 241–250 (1999)
10. Jain, M., Dovrolis, C.: End-to-end available bandwidth: measurement methodology, dynamics, and relation with TCP throughput. IEEE/ACM Trans. Netw. 11(4), 537–549 (2003)

11. Dovrolis, C., Ramanathan, P., Moore, D.: Packet-dispersion techniques and acapacity-estimation methodology. IEEE/ACM Trans. Netw. 12(6), 963–977 (2004)
12. Prokkola, J., Hanski, M., Jurvansuu, M., Immonen, M.: Measuring WCDMA and HSDPA Delay Characteristics with QoSMeT. In: Proceedings of the IEEE International Conference on Communications, pp. 492–498 (June 2007)
13. Saroiu, S., Gummadi, K.P., Gribble, S.D.: SProbe: A fast tool for measuring bottleneck bandwidth in uncooperative environments (2002), http://sprobe.cs.washington.edu
14. Savage, S.: Sting: A TCP-based network measurement tool. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems, pp. 71–79 (1999)
15. Augustin, B., Cuvellier, X., Orgogozo, B., Viger, F., Friedman, T., Latapy, M., Magnien, C., Teixeira, R.: Avoiding traceroute anomalies with paris traceroute. In: Proceedings of the ACM Internet Measurement Conference, pp. 153–158 (2006)
16. Abela, J., Debeaupuis, T.: Universal format for logger messages (1999), http://tools.ietf.org/id/draft-abela-ulm-05.txt
17. Open Log Format, The OLF standard (2009), http://www.openlogformat.org/
18. Paxson, V., Mahdavi, J., Adams, A., Mathis, M.: An architecture for large-scaleInternet measurement. IEEE Comm. Magazine 36(8), 48–54 (1998)
19. Magana, E., Morato, D., Izal, M., Aracil, J., Naranjo, F., Astiz, F., Alonso, U., Csabai, I., Haga, P., Simon, G., Steger, J., Vattay, G.: The European traffic observatory measurement infrastructure (ETOMIC). In: Proceedings IEEE Workshop on IP Operations and Management (October 2004)
20. DIMES, The DIMES project, (2009), http://www.netdimes.org/
21. Anderson, D.P., Cobb, J., Korpela, E., Lebofsky, M., Werthimer, D.: Seti@home:an experiment in public-resource computing. Comm. ACM 45(11), 56–61 (2002)
22. Danalis, A., Drovolis, C.: Anemos: An autonomous network monitoring sys- tem. In: Proc. of 4th Passive and Active Measurements Workshop, San Diego, CA (2003)
23. Johnson, D., Gebhardt, D., Lepreau, J.: Towards a high quality path-oriented network measurement and storage system. In: Claypool, M., Uhlig, S. (eds.) PAM 2008. LNCS, vol. 4979, pp. 102–111. Springer, Heidelberg (2008)
24. Krishnamurthy, B., Madhyastha, H.V., Spatscheck, O.: ATMEN: a triggered network measurement infrastructure. In: Proceedings of the 14th ACM International Conference on World Wide Web, New York, USA, p. 499 (2005)
25. Song, H.H., Qiu, L., Zhang, Y.: NetQuest: a flexible framework for large-scalenetwork measurement. IEEE/ACM Trans. Netw. 17(1), 106–119 (2009)
26. Spring, N., Wetherall, D., Anderson, T.: Scriptroute: a public internet measurement facility. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems, pp. 17 (2003)
27. Saint-Andre, P.: Extensible Messaging and Presence Protocol (XMPP): Core. RFC 3920 (2004)
28. Moura, A.L.D., Ierusalimschy, R.: Revisiting coroutines. ACM Trans. Pro- gram. Lang. Syst. 31(2), 1–31 (2009)
29. CAIDA, Archipelago measurement infrastructure (2009), http://www.caida.org/projects/ark/