The network neutrality bot architecture: a preliminary approach for self-monitoring of Internet access QoS

Simone Basso*, Antonio Servetti†, Juan Carlos De Martin*
*NEXA Center for Internet & Society
Dipartimento di Automatica e Informatica
Politecnico di Torino
†Dipartimento di Automatica e Informatica
Politecnico di Torino

***Abstract*—The "network neutrality bot" (Neubot) is an evolving software architecture for distributed Internet access quality and network neutrality measurements. The core of this architecture is an open-source agent that ordinary users may install on their computers to gain a deeper understanding of their Internet connections. The agent periodically monitors the quality of service provided to the user, running background active transmission tests that emulate different application-level protocols. The results are then collected on a central server and made publicly available to allow constant monitoring of the state of the Internet by interested parties.**

In this article we describe how we enhanced Neubot architecture both to deploy a distributed broadband speed test and to allow the development of plug-in transmission tests. In addition, we start a preliminary discussion on the results we have collected in the first three months after the first public release of the software.

I. Introduction

The debate on "network neutrality" is becoming a more and more relevant topic in economic, technical and even political environments [1]. The basic question is whether network operators should be allowed to differentiate the Internet traffic that goes through their infrastructure or whether network neutrality should be explicitly safeguarded by the law, thereby enshrining what has been a characteristic of the Internet since its birth.

The ability to block or slow down the traffic, which can be used to prevent the spreading of spam, viruses, botnets, and other malwares, can also be used by Internet Service Providers to implement very questionable policies [2]. Apart from the so-called "Great Firewall of China" and other censorship efforts, which are beyond the scope of this paper, differentiating technologies can also be employed to throttle: (i) the "seeding" file-sharing traffic that flows out of ISPs networks (traffic considered "bad" because often providers are charged per-Megabyte for the traffic exchanged with their upstream Internet Providers), (ii) the file-sharing traffic generated during rush hours (in an effort to avoid the collapse of under-provisioned access networks), (iii) the traffic of some "over-the-top" (OTT), possibly free, services that compete with "managed services" that an ISP sells (for example Skype competing with ISP own Voice-over-IP solution). In particular, the conflict between managed and OTT services, e.g. YouTube, Skype, is becoming more and more relevant (particularly in the US). Providers (a) have started to offer additional managed services along with the Internet connection, such as television, video, and voice communication, and (b) often employ differentiating technologies, and other practices such as *bandwidth caps*, to guarantee that there is always enough bandwidth to carry the managed services (while OTT services get the traditional "best effort" treatment) [3].

Technically, with the advent of "deep packet inspection" and other classification technologies, network differentiation is a two-step process. Filtering is performed to classify packets at the edge of the Provider's network. Then, inside the network, packets receive the service level associated with their class. In particular, packets that belong to low priority classes might be: (i) diverted on slower and/or more congested "virtual links", with traffic engineering technologies like MPLS (Multiprotocol Label Switching), (ii) dropped from the router queues in cases of congestion, (iii) scheduled for forwarding after higher priority packets.

Despite the potential impact of packet filtering, users, developers, and most other network administrators are not provided with enough information regarding these practices, and there are no well-established tools to assess whether a particular kind of discrimination is active on a given connection. As a consequence, we are designing Neubot as a software architecture for distributed network measurements [4] on which developers can implement tools and methodologies that will help examining ISPs throttling of Internet traffic depending on the protocol used and service offered. In addition, Neubot results will be made publicly available at `neubot.org` [5] for further analysis by the research community, and to inform the international debate on network neutrality with real, per-host, network measurements.

The rest of this paper is organized as follows. In section II, we introduce the architecture and we describe the proposed measurement techniques. In section III, we describe the implementation of our broadband speed test and the enhancements

that allow plug-in transmission tests. In section IV, we start a preliminary discussion of the collected results. In section V, we compare Neubot with related work. Finally, conclusions are drawn in section VI.

II. Architecture

The architecture of Neubot consists of an *agent* that volunteer end-users shall install on their computers, and a set of servers. There is a *master server*, `master.neubot.org`, that coordinates the tests and collects the results. And there are *test servers* that implement one or more transmission tests.

The agent runs in background. Under Linux, BSD, and other Unixes the agent is started at boot time, becomes a daemon, and drops root privileges, running on behalf of the restricted user `_neubot`. Under Windows the agent is started when a user logs in, and runs in the context of her session.

The agent listens on port `9774` of the local host and implements a *JSON API*. The user can control the agent using a *web user interface*, based on this API, that also allows to review recent results. Given the flexibility of the JSON API, other interfaces are possible.

The agent automatically performs a set of transmission tests between the Neubot computer and one or more test servers (client-server mode), and between the Neubot computer and other Neubots (peer-to-peer mode). Periodically, test results are archived, sending them back to the master server.

A. Client-server test description

The diagram in Fig. 1 shows the sequence diagram and components involved in a generic test. The *Database* and *Coordinator* components are installed on the master server. In the case of a client-server test, the *TestNegotiator* and *TestProvider* are installed on a test server. Note that it's possible for the master server and the test server to be the same physical (or virtual) machine, for simplicity and maintainability. Each component plays a well-defined role, as explained below.

The Coordinator keeps track of all the available test servers and *test peers* – i.e. agents running in peer to peer mode and accepting connections, as explained in section II-B – and *binds* an agent that wants to perform a test with the test server (or test peer) implementing such test.

The TestNegotiator assigns a temporary unique identifier to each connecting agent that wants to perform a transmission test, and *manages* the queue of incoming tests, to make sure that tests are always well provisioned. At the end of the test, the TestNegotiator collects and stores the test results on the local database.

The TestProvider is the component that uses a given protocol to implement a certain transmission test to estimate selected characteristics of the network between the agent and the TestProvider itself. Note that we present the TestNegotiator and the TestProvider separately, because they are different components, but, in the current source tree, they are always implemented together, for the sake of simplicity.

The Database is the component that collects all the results of all the tests that have been performed. This could either be a single server or, possibly, a set of distributed servers.

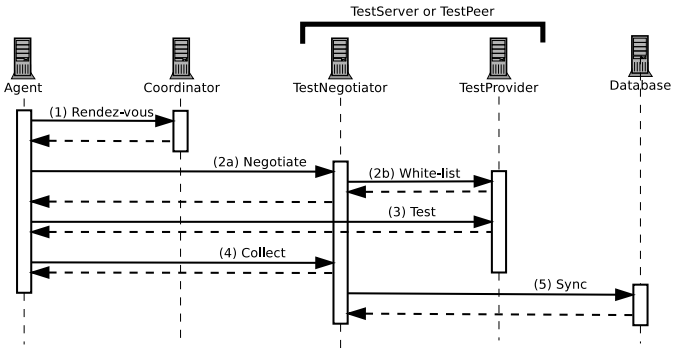

Fig. 1. Sequence diagram of a generic test.

At the beginning, the Coordinator knows in advance a list of well-known test servers. So, when an agent connects (1) to get the address of a test server, the Connector could return a list of one or more addresses. The fact that the Coordinator populates the list of tests allows for very flexible probing, because different tests could be returned depending on the circumstances.

Given the address of a TestNegotiator, the agent negotiates the permission to perform the test (2a), possibly proposing certain test parameters. The negotiator assigns the agent a random and unique identifier, informs the agent on its position in queue, and possibly returns the negotiated parameters. When the TestProvider load allows to unblock the client (2b), the TestNegotiator informs the client that now it can perform the transmission test, and starts a per-test timer. If such timer expires, the negotiator assumes that the test is taking too long, possibly because the agent computer was shut down, and the temporary unique identifier is removed from the white list[1].

Then there is the transmission test between the agent and the TestProvider (3). The measured performance metrics depends on the target protocol, but always include the hosting computer load, i.e. network, CPU, and memory usage. An HTTP test inspired by Speedtest.net test [6] is already implemented. We are working on a BitTorrent test inspired by Glasnost [7]. Other target protocols include RTP, the IETF Real-Time Transport Protocol, used by many Internet-based multimedia applications, and Skype's proprietary peer-to-peer Voice-over-IP protocol.

Once the test is completed, the agent uploads the results (4) to the TestNegotiator. The latter, in turn, stops the per-client timer and saves results in the local database. When a significant batch of results has been collected, the TestNegotiator will upload it upstream to the master Database (5).

Finally, the agent goes idle for a long amount of time (at least fifteen minutes) before repeating again the procedure explained above.

B. Peer-to-peer test description

In future, it will be possible for Neubot agents to listen for incoming connections and implement the "server side"

[1]Of course, the TestProvider aborts the connection if the connecting agent is not in the white list.

of a test, i.e. the TestNegotiator and the TestProvider. The difference is that, when an agent performs the rendezvous (1), it registers with the Coordinator as a *listener* rather than as a *connector* for a given protocol. The Coordinator would then check whether the agent is not firewalled, and, if the check succeeds, it will add the agent Internet address to its list of available test negotiators and providers. The remainder of the procedure is exactly as in the client-server test, with the notable difference that now the connecting agent connects to another agent and not to an ad-hoc test server.

III. Implementation

In this section we provide more details regarding Neubot modules, discuss briefly how the implemented tests works, and describe the deployment of Neubot. This discussion builds on the more generic explanation of the required modules, available in our previous work [8].

A. Modules

Neubot consists of: (i) an executable *startup script*, written in Python, or, under Windows, a `PE/COFF` executable, frozen using py2exe [9]; (ii) a *library of modules*, written for Python 2.5-2.7, implementing the agent and the server; (iii) a *web user interface* written in Javascript using jQuery [10]; and (iv) a *Gtk+ status icon* written in PyGtk [11], available under GNU/Linux, and other Unixes.

The design of Neubot is module oriented. Most Neubot modules export a function, named `main`, that parses command line options and implements some behavior. The startup script acts like a switch. It maps the first command line option to the related module and then dispatches the control to this module. So, depending on the first command line option, the startup script might start the Neubot agent in background, run a background process that hosts any of the server side components described in section II-A, run a given test from command line, open the web user interface in the default browser, or show the status icon in the notification area.

New tests could be added as plugins. For example, the diagram in Fig. 2 shows the dependencies of a new hypothetical transmission test named *proto*. These dependencies are: (i) the *protocol stream*, on top of Neubot generic stream module; (ii) the *protocol test* itself, depending on the marshal module as well as on (i); (iii) a set of *javascript protocol helpers* for the web user interface to represent the results and manage the test parameters.

The *http stream* module, used to implement the transmission test described in section III-B, is also employed to send the "Neubot to Neubot" messages that implement the *rendezvous*, *negotiate*, and *collect* transactions described in section II-A. Such messages are encoded in (and decoded from) either `JSON` or `XML` using the services provided by the marshal module.

We now describe in detail the streams, marshal, and libneubot.js modules of the Neubot library.

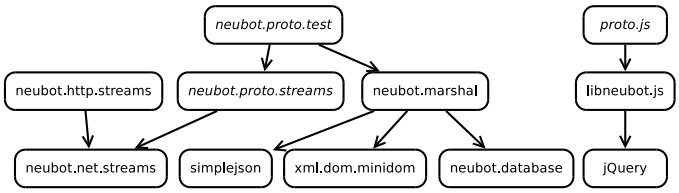

Fig. 2. This figure shows the dependencies of a new hypothetical transmission test module named *proto*.

1) Streams: This module is built on-top of "asynchronous event-based I/O", implemented using `select()`. We use this I/O strategy because it allows to handle multiple concurrent small XML or JSON "Neubot to Neubot" messages without consuming too much resources and because it enables better performances. In particular, there is little overhead in receiving many small messages – we don't need to spawn extra processes or threads, or manage thread pools – and the process should never block waiting for I/O to complete.

This module provides consumer modules several facilities, such as support for: scrambling messages using the RC4 implementation provided by PyCrypto [12]; full encryption and authentication using the Secure Socket Layer API available since Python 2.6 [13]; support for enabling "time to connect" and throughput measurements, eventually creating groups of connections.

In addition, this module enforces a strict event model for implementing protocols on top of the generic stream. Such a model is inspired by the "protocol" model of Twisted [14] and, to some extent, by the "handler" model of BitTorrent mainline [15]. This simplifies the task of writing new transmission tests for Neubot.

2) Marshal: The marshal module provides mechanisms to marshal and unmarshal *simple classes*, i.e. classes that contains just *simple types* – integers, (unicode) strings, floating point numbers – as well as vector and/or dictionaries of simple types. This is almost automatic with simplejson [16], while for XML we have written wrappers around Python Document Object Model (DOM) that perform this task.

This module also implements helper code to generate the queries to create per-test tables and to save data, given both the name of the table and a prototype of the object that should be saved into the table. This simplifies the task of writing new transmission tests for Neubot.

3) libneubot.js: This module is a collection of helpers and wrappers to retrieve data using the "Neubot web api", i.e. the JSON-based API to query the local Neubot daemon from the web interface, or from other clients, such as the Gtk+ status icon.

B. Speedtest

This transmission test is inspired by the online test available at Speedtest.net [6]. Here we don't describe the *negotiate* and *collect* transactions, because they have already been described in section II-A, and we just focus on the test.

This test employs two connections to increase throughput with moderately congested *long fat networks*. We decided to use two connections even if in literature it's common to suggest to use four connections (see Tierney [17]) because our goal is not to maximize the throughput at the expense of other connections but rather to get an estimate of the available bandwidth.

We are now going to discuss the three steps that compose this test: the evaluation of the round-trip time latency, and the estimation of the bandwidth available in the downstream and upstream paths.

1) Latency: The first step consists in the evaluation of the round-trip time (RTT) latency between the Neubot agent and the server. We implement two techniques to do that: (i) "time to connect" that estimates the round-trip time using the time required for `connect()` to complete; and (ii) "short HTTP transaction" that estimates the RTT using the time elapsed between sending a small `HEAD` request, and receiving the response – which, per RFC2616 [18], should consist of headers only. The former technique collects a sample per connection, while the latter allows to collect an arbitrary number of samples per connection. The former technique should yield a better estimate because it has no application-level overhead, given that the three-way handshake is entirely performed by the operating system kernel.

2) Download: The second step consists in the estimation of the available downstream bandwidth. To do that, it requests a portion of an huge resource and doubles the portion size until the download takes a significant amount of time[2]. The current implementation gives up when the test takes more than a second to complete – not to disrupt the user experience for too much time. We are refining the code to tune the download interval so that it spans an integral amount of round-trip times. Another forthcoming enhancement is to auto tune the TCP buffer, given the estimated RTT and the bandwidth we expect, either from guessing or by inspecting the download history. These adjustment should make the test more robust for connections with very high *bandwidth-delay product*.

3) Upload: The third step is similar to the second one, but it targets the upstream bandwidth. This step uses `POST`, doubling the uploaded portion size until it takes more than one second. The content we push upstream is a portion of the data that has been downloaded during step two (the assumption is that the upstream speed will not be greater than the downstream one).

C. Deployment

We deployed our main test server at Torino-Piemonte Internet eXchange (TOP-IX), the nearest Internet eXchange (IX) to our campus, on a 1 GByte / 2 GHz / 1 core Debian 5.0 virtual machine, attached to a 1 Gbit/s upstream/downstream pipe. We decided to deploy Neubot in the network of an IX in order to minimize the "distance" between the installed Neubot agents and our main server.

The virtual machine at TOP-IX runs a rendezvous and a speedtest server, using two different processes, to allow to reload the list of known speedtest servers without stopping the local speedtest server instance. Indeed, in a couple of cases, we added one more speedtest server, located at Politecnico di Torino, for the purpose of redirecting there half of the traffic, to test experimental algorithms.

[2]We use `HTTP Range` header to implement that.

IV. Preliminary results

In this section we present some preliminary results of the *speedtest* test, extracted from our database that contains tests since October, 3rd 2010 until January, 17th 2011, counting up to 448317 tests and 1115 unique agents.

We focus on download speed because broadband connections are nearly always advertised "up to" a certain download speed. Therefore, we are interested to study the difference between the advertised speed and the measured speed when downloading from a well-known nearby location in the network (in our case, the server at TOP-IX).

We identify a reasonable *goodset* of the collected results, show the distribution of download speed in that set, compute the average speed and compare it with the one reported by Speedtest.net [6] and Youtube [19]. Then, we select a Neubot agent and we show the download speed distribution of the tests performed by such agent. The agent choice is not random. We choose an agent whose results are consistently within the goodset.

In the goodset, we consider only tests where the measured download speed was lower than 20 Mbit/s and the estimated RTT latency[3] was lower than 100 ms. Overall, this goodset includes up to 338615 tests, i.e. 75% of the total number of tests. And the tests in the goodset have been performed by 1035 distinct agent identifiers (94% of the total).

The restriction on the download speed is to single out tests performed from ADSL connections. There were many high speed results due to tests originating from our campus or from TOP-IX, and the best rated download speed for consumer ADSL in Italy is indeed 20 Mbit/s.

The restriction on the RTT latency removes results collected from locations "too far away" from our server at TOP-IX, where the outcomes are less significant, mainly due to high bandwidth-delay product, and possibly due to the presence of cross traffic and congestion.

A. Aggregate download speed distribution

The plot in Fig. 3 shows the download speed distribution of a *goodset* of the collected results. From the analysis of this data, we clearly see peaks before 1, 2, and 7 Mbit/s, which are common ADSL speeds in Italy.

B. Average download speed distribution

We calculated the average download speed distribution in the goodset, and we compared the average with the one provided by Speedtest.net [6] and Youtube [19]. We do not show the variance because that makes little sense, for results

[3]To estimate RTT we use the "short HTTP transaction" method only, because, with certain versions of Neubot, "time to connect" mistakenly takes DNS resolution into account.

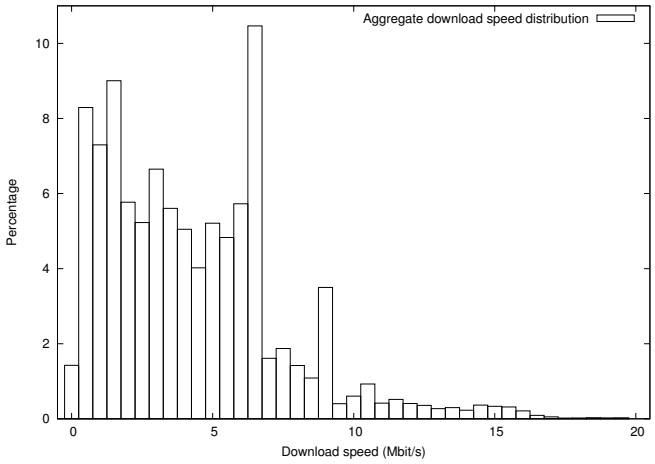

Fig. 3. Aggregate download speed distribution.

collected from different connections with different access speeds.

We compare the goodset average with Speedtest.net's [6] average download speed for Italy. In particular, we pick the average that excludes universities and companies. This is reasonable because we removed very high-speed clients from the goodset and universities and companies often enjoy high-speed connectivity. With respect to Youtube Video Speed History [19], we use the average for Turin rather than the average for Italy because the goodset contains clients "not too far" from Turin, due to the 100 ms RTT restriction.

The results are shown in table I.

C. Single-agent download speed distribution

In this subsection we comment on the distribution of the download speed of a Neubot agent we have selected.

The selected agent's tests consistently fall within the goodset. It has performed 1383 tests and the RTT latency falls 84% of the time within 100 ms. Moreover, it appears the be attached to a 7 Mbit/s ADSL connection. Indeed, a simple `whois` lookup shows that the IP addresses employed by such agent belong to an Internet Service Provider whose top offer features 7 Mbit/s downstream. Furthermore, the agent's download speed is always lower than 7 Mbit/s.

The diagram in Fig. 4 shows the distribution of the download speeds measured by the selected agent. Note that we extracted the results of this agent from the database, given that each agent is identified by a random `UUID`. But, since results are also saved locally, in principle the owner of the selected agent could have done the same analysis, accessing the local database and using custom scripts, or, possibly via the Neubot agent's web interface.

Note that most of the results in Fig. 4 are distributed in proximity of 7 Mbit/s, but there are significant spikes near 4 Mbit/s and 1 Mbit/s too (two other common downstream speeds for ADSL in Italy). We are currently investigating the causes of the low speed spikes, and checking whether performing longer tests would yield more stable results.

Source	Download speed (Mbit/s)
Neubot	4.58
Speedtest.net	4.68
YouTube	3.03

TABLE I
AVERAGE RESULTS OF NEUBOT COMPARED WITH AVERAGE RESULTS OF SPEEDTEST.NET IN ITALY AND YOUTUBE VIDEO DOWNLOAD HISTORY FOR TURIN.

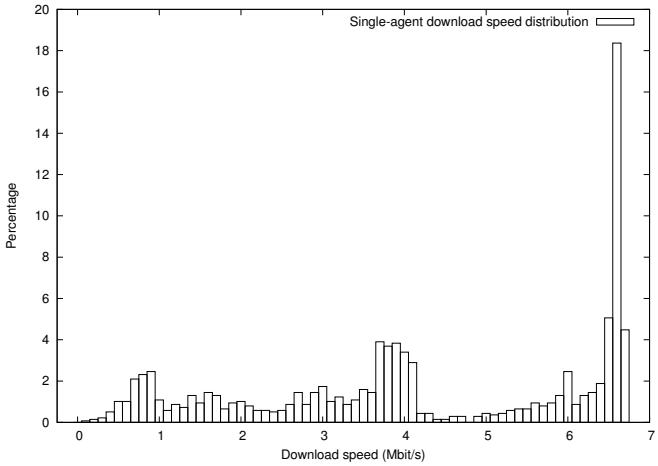

Fig. 4. Single-agent download speed distribution.

V. RELATED WORK

Most of the literature devoted to network neutrality focuses on testing only one specific kind of disruption. For example, Diffprobe tries to infer whether there is protocol-dependent shaping [20]. Worth mentioning also is: (i) Weaver, Sommer, and Paxson's paper, that provides a rich set of heuristics to detect spoofed RST segments and identify, with a certain degree of confidence, the generating device [21]; (ii) the Glasnost project, that provides a user-friendly Java applet that performs a BitTorrent-versus-random-data test trying to detect whether the ISP blocks BitTorrent traffic [7]; (iii) Zhang, Mao, and Zhang's paper, that studies the amount of traffic differentiation in the backbone, employing a protocol-aware traceroute-like tool [22].

Like Neubot, NANO [23] and Grenouille [24] deploy more general approaches that are able to quantify a broad range of network neutrality violations. NANO (Network Access Neutrality Observatory) employs passive measurements. The client continuously monitors user-generated traffic, and periodically sends throughput, round-trip-time, and other general performance metrics to the server, as well as ancillary information including the state of the hosting computer, its geographic location, the browser, and the operating system. Interestingly, the server relies on stratification to cluster the clients in strata where the difference in performance depends only on the fact that different ISPs have employed different policies. Grenouille has an architecture similar to Neubot, but a slightly different goal: to measure ISP backbone congestion. Every

30 minutes, the client connects to a server standing near the edge of the ISP network, to avoid traversing (many) other ISPs networks. This way, the FTP upload, FTP download, and ping tests are not (much) biased by other ISPs, and hence it is possible to evaluate the average quality of service. To avoid overloading the servers, each client displaces tests in time by a random amount of seconds. Tests results are reported to a central server, unless the client finds that the user has consumed too many network resources during the test. The central server analyzes the results, producing daily and monthly, global and per-city charts.

VI. Conclusion and Future Work

In this paper we presented the Neubot architecture for distributed network neutrality measurements. In comparison with the other known approaches, the value of Neubot lies in its ability to permanently monitor the end-user Internet connection instead of performing discrete probes. Such results, made publicly available, will allow a systematic analysis of Internet services together with a deeper understanding of network neutrality based on real, per-host, network measurements. In addition, as far as we know, the ability to perform distributed test will give birth to the first P2P overlay for active network measurements.

In this early stage of Neubot we implemented a broadband speed test and we collected a total of 448317 tests from 1115 unique agents in the first three months since the first public release. The purpose of this initial phase was to validate the architecture robustness and the effectiveness of the transmission test, using a "testbed" transmission protocol like HTTP. So, we compared the results with other, well known, online sources, such as Speedtest.net [6], and YouTube Video Download History [19]. The preliminary analysis of the collected data set shows that the average download speed is comparable with what other services report, provided that we make reasonable assumptions to single out tests performed by ADSL connections that were not too far away from Neubot test server.

We have also described ongoing efforts to enhance the implementation, by easing the integration of diverse transmission tests that target other protocols. These efforts will help estimate differential treatments. The enhancements include a stricter protocol model, support for marshaling and unmarshaling test results, using both JSON/XML and SQL, and helpers for the web user interface.

The current implementation of Neubot will be extended to allow for distributed measurements with the inclusion of a client-server test to detect BitTorrent discrimination. From then on, we plan to focus our research on the analysis of the collected data with particular attention to the relation between the measurements of the Internet Service Provider and the geographic location of the Neubot agent that performed the tests.

Acknowledgment

We would like to thank Prof. Jean-Claude Guédon and Dr. Federico Morando for their support and for making suggestions to improve the readability of this paper.

References

[1] J. Crowcroft, "Net neutrality: the technical side of the debate: a white paper," *ACM SIGCOMM Computer Communication Review*, vol. 37, no. 1, pp. 49–56, 2007.

[2] S. Jordan, "Some traffic management practices are unreasonable," in *Computer Communications and Networks, 2009. ICCCN 2009. Proceedings of 18th Internatonal Conference on*. IEEE, 2009, pp. 1–6.

[3] R. Ma, D. Chiu, J. Lui, V. Misra, and D. Rubenstein, "On cooperative settlement between content, transit and eyeball internet service providers," in *Proceedings of the 2008 ACM CoNEXT Conference*. ACM, 2008, pp. 1–12.

[4] J. De Martin and A. Glorioso, "The Neubot project: A collaborative approach to measuring internet neutrality," in *Technology and Society, 2008. ISTAS 2008. IEEE International Symposium on*. IEEE, 2008, pp. 1–4.

[5] Neubot, the network neutrality bot. [Online]. Available: http://www.neubot.org/

[6] Speedtest.net - The Global Broadband Speed Test. [Online]. Available: http://speedtest.net/

[7] M. Dischinger, A. Mislove, A. Haeberlen, and K. Gummadi, "Detecting bittorrent blocking," in *Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*. ACM, 2008, pp. 3–8.

[8] S. Basso, A. Servetti, and J. De Martin, "Rationale, Design, and Implementation of the Network Neutrality Bot." [Online]. Available: http://www.neubot.org/neubotfiles/aica2010-neubot-paper.pdf

[9] FrontPage - py2exe.org. [Online]. Available: http://www.py2exe.org/

[10] jQuery: The Write Less, Do More, JavaScript Library. [Online]. Available: http://jquery.com/

[11] PyGtk: GTK+ for Python. [Online]. Available: http://www.pygtk.org/

[12] PyCrypto – The Python Cryptography Toolkit. [Online]. Available: http://www.dlitz.net/software/pycrypto/

[13] 17.3. ssl – SSL wrapper for socket objects. [Online]. Available: http://docs.python.org/release/2.6.6/library/ssl.html

[14] Twisted Matrix Labs – Building the engine of your internet. [Online]. Available: http://twistedmatrix.com/trac/

[15] BitTorrent / open source. [Online]. Available: http://www.bittorrent.com/opensource

[16] Simple, fast, extensible JSON encoder/decoder for Python. [Online]. Available: http://pypi.python.org/pypi/simplejson/

[17] B. Tierney, "TCP tuning guide for distributed application on wide area networks," *USENIX & SAGE Login*, vol. 26, no. 1, pp. 33–39, 2001.

[18] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "RFC2616: Hypertext Transfer Protocol–HTTP/1.1," *RFC Editor United States*, 1999.

[19] Youtube Video Speed History. [Online]. Available: http://www.youtube.com/my_speed

[20] P. Kanuparthy and C. Dovrolis, "Diffprobe: detecting ISP service discrimination," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.

[21] N. Weaver, R. Sommer, and V. Paxson, "Detecting forged TCP reset packets," in *In Proc. of NDSS*. Citeseer, 2009.

[22] Y. Zhang, Z. Mao, and M. Zhang, "Detecting traffic differentiation in backbone ISPs with NetPolice," in *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*. ACM, 2009, pp. 103–115.

[23] M. Tariq, M. Motiwala, N. Feamster, and M. Ammar, "Detecting network neutrality violations with causal inference," in *Proceedings of the 5th international conference on Emerging networking experiments and technologies*. ACM, 2009, pp. 289–300.

[24] Grenouille.com - la météo du net depuis 2000. [Online]. Available: http://www.grenouille.com/