POLITECNICO DI TORINO

SCUOLA DI DOTTORATO

Dottorato in Ingegneria Informatica e dei Sistemi – XXVI ciclo
ING-INF05/Sistemi di Elaborazione delle Informazioni

Tesi di Dottorato

Neubot: A Software Tool Performing Distributed Network Measurements to Increase Network Transparency

`http://nexa.polito.it/neubot`

Simone Basso

Tutore
prof. Juan Carlos De Martin

Coordinatore del corso di dottorato
prof. Pietro Laface

Luglio 2014

Summary

We present Neubot (the network neutrality bot), a network-measurement platform designed to run network-performance experiments from the network edges. The data produced by Neubot is useful to increase network transparency and to study network neutrality. We describe the Neubot architecture (based on plugins that emulate several protocols and are able to run client-server and peer-to-peer tests), which is one of the main contributions of this thesis. We describe the current Neubot implementation (Neubot 0.4.16.9), we provide up-to-date data concerning Neubot deployment, and we show how we used Neubot to run four diverse large-scale measurements campaigns involving more than 1,000 Neubot instances each. Such measurements campaign, which were only possible because the Neubot architecture was already flexible enough to allow us to deploy new network experiments on the already installed Neubot instances, were concerned with, respectively: the measurement of broadband speed using the HTTP protocol; the study of the link between application-level measurements and the packet-loss rate experienced by TCP (which is the other main contribution of this thesis); the study of rate adaptation algorithms for the dynamic adaptive streaming over HTTP streaming technology (DASH); emulating the BitTorrent protocol. We conclude the thesis with the description of Neuviz (the Neubot visualizer), a prototype data-visualization architecture that loads Neubot data and allows to navigate the data looking for potential deviations from network neutrality. Despite being still in beta stage, Neuviz already allowed to spot three anomalies in the median speeds measured by the Neubot 'HTTP Speedtest' and BitTorrent tests.

Contents

Chapter 1

Introduction

Network neutrality is the principle that all Internet packets shall be treated equally. The network-neutrality principle (which is rooted into the original Internet design principles) has recently been endangered by techniques that allow an Internet service provider (ISP) to discriminate the Internet packets. This fact generated a vibrant network-neutrality debate in academic, policy and news circles: according to many scholars, in fact, the network neutrality principle is key to guarantee many of the economically- and socially-desirable properties of the Internet (e.g., the possibility to innovate without asking the permission to anyone and the possibility for anyone to produce content).

The network-neutrality debate is extraordinarily vast. It encompasses many disciplines and it may be framed according to a number of diverse point of views (e.g., fundamental rights, competition law, consumer rights, Internet-performance measurements). We provide the reader a glimpse of the complexity of the topic, then we dive into the information-and-communication-technology (ICT) perspective (see Section 1.1). We describe which techniques may be used to discriminate Internet packets, showing that discrimination is correlated with degraded performance (see Section 1.2). We provide examples of violations of the network neutrality principle (see Section 1.3). We provide a tour of network-neutrality solutions (see Section 1.4). Finally, we conclude the Chapter with a description of the structure of this thesis that highlights the main contributions provided therein (see Section 1.5).

1.1 A Definition of Network Neutrality

According to the 'Network Neutrality FAQ' [116], maintained by Tim Wu, the Columbia law school professor who coined the term 'network neutrality':

> Network neutrality is best defined as a network design principle. The idea is that a maximally useful public information network aspires to treat all content, sites, and platforms equally. This allows the network to carry every form of information and support every kind of application. The principle suggests that information networks are often more valuable when they are less specialized – when they are a platform for multiple uses, present and future.

The network neutrality principle – argues Tim Wu – is very similar to the 'end-to-end' design principle, which is one of the cornerstones of the Internet. Described in a 1981 paper by Saltzer, Reed and Clark for the 2nd IEEE International Conference on Distributed Systems later adapted for the ACM Transactions on Computer Systems [277], the end-to-end principle (or argument) states that, in a layered system, a functionality shall be closer to the application that uses it, unless there is a need to place such functionality at a lower-level for performance reasons.

As far as the ancient Internet was concerned, in particular, the end-to-end design principle called for a relatively thin IP layer and for intelligence implemented from the TCP layer upwards. Such design choice allowed for fast packet switching, which otherwise at the time would not have been possible, given the scarce processing power of routers. However, such choice was not only merely instrumental to bootstrap the early Internet. Rather, the end-to-end principle became so integral to the Internet culture (as opposed to the telco culture) that, nowadays, Bram Cohen (author of BitTorrent [27]), speaking of where to implement innovative functionality (e.g., protocols that avoid creating too much queue at the bottleneck), affirms that "it's best to do everything with smarts at the end points rather than in the router though, because it's more deployable and flexible" [197].

As we said, there was a period of time during which routers had not enough power to efficiently process the packets payload. However, starting from the early 2000s, gradually routers become more and more able to recognize and discriminate diverse protocols and applications, using a variety of techniques, on which we will return in the next Section. At the same time, the structure of the Internet market changed as well. While at the time of voiceband modems users could call (using landline telephone) several Internet providers, from the deployment of the ADSLs onward such flexibility was lost. In the 2000s the norm was that a single ISP/telco provided both the last mile (i.e., the portion of the network that joins the user's premise and the first Internet router) and Internet access [281].

Network neutrality proponents started to fear that ISPs – controlling the last mile and having the technology to discriminate the traffic – could create artificial scarcity to favor their own services over services provided by third parties. Arguments in favor of network neutrality touch issues ranging from freedom of expression, to market structure (as discussed above), from competition to consumer rights. Many proponents believe that the network neutrality is key to preserve the non-discriminatory Internet ecosystem in which (a) everyone is free to innovate without asking the permission and (b) anyone is free to produce content. Scholars that expressed themselves in favor of network neutrality include: Tim Wu [304], Mark Lemley and Lawrence Lessig [240], Brett Frischmann [216], Barbara van Schewick [299] [300], Jonathan Zittrain [307] and Chris Marsden [244].

Opponents of network neutrality argue that ISPs need to provide some kind of data discrimination to finance the creation of the so-called fiber-optics-based next-generation networks (NGN). For example, John Thorne, a Verizon senior vice president and deputy general counsel, said to the Washington Post [255]:

> The network builders are spending a fortune constructing and maintaining the networks that Google intends to ride on with nothing but cheap servers [...]

> It is enjoying a free lunch that should, by any rational account, be the lunch of the facilities providers [...] The only way we are going to attract the truly huge amounts of capital needed to build out these [NGN] networks is to strike down governmental entry barriers and allow providers to realize profits.

Other opponents of network neutrality contend that, once the NGN networks will be built, there will be enough capacity and no network neutrality issues. In this vein, Bret Swanson (president of Entropy Economics LLC, a strategic insight firm specializing in technology) argued in the Wall Street Journal [289] [290]:

> Ironically, the condition that net neutrality seeks to ban – discrimination or favoritism of content on the Internet – is only necessary in narrowband networks. When resources are scarce, the highest bidder can exclude the others. But with real broadband networks, capacity is abundant and discrimination unnecessary. Net neutrality's rules, price controls and litigation would prevent broadband networks from being built, limit the amount of available bandwidth and thus encourage the zero-sum discrimination supposedly deplored.

Between proponents and opponents, finally, there are those who argue that the network-neutrality topic is complex and nuanced, especially from the technical side. For example, if we ban the technologies that allow ISPs to inspect the content of the packets – says David Farber (Distinguished Career Professor of Computer Science and Public Policy at the School of Computer Science, Heinz College, and Department of Engineering and Public Policy at Carnegie Mellon University) [229] – we also prevent ISPs from automatically detecting botnets, attacks and spam. Similarly, adds David Clark (Senior Research Scientist at MIT's Computer Science and Artificial Intelligence Laboratory), the discussion on privacy is more complex than lets-stop-ISPs-from-looking-at-our-traffic: in 2009 he argued that "[w]hat we are going to be fighting about in two years is who has the right to observe everything you do" [229], ISPs (using the traffic inspection and discrimination technologies) or Internet giants such as Facebook and Google. Along these lines of complexity, many skeptics of network neutrality fear that a rigid legislation on network neutrality may be unfortunate, hampering the whole Internet ecosystem. For example, in 2006 Bram Cohen (author of BitTorrent) argued that it was not obvious how to create a network neutrality legislation that would allow ISPs to protect users from spam, botnet and attacks [242].

Although we stop this brief tour of the neutrality debate here, we hope to have given the reader a glimpse of the complexity of the topic and of the many available points of view (competition, market structure, fundamental rights, etc). We suggest as reference for further readings on the topic the work by Jon Crowcroft [199], the work by Krämer, et al., [235] and the Wikipedia entry on network neutrality [159].

As a final comment, let us explain why transparency is key to understand network neutrality. There is currently a profound information asymmetry between users and ISPs. ISPs and content providers, in fact, have access to a wealth of information, because they control and monitor their own infrastructure. Users and applications, instead, live in the dark. The few information to which they could theoretically have access (e.g., bandwidth,

latency and losses) is hidden behind the opaque TCP/IP stack, which was designed at times in which the network was not supposed to discriminate. Therefore, more transparency is needed, in form of measurement techniques, tools and data, to shed light on what happens to the user flows and, in turn, to rebalance the information asymmetry.

From now on, we will progressively narrow our point of view. We are about to discuss how routers can discriminate among flows. Next we will move the focus to network neutrality solutions, talking about the particular solution discussed in this thesis: measurements from the edges, performed with Neubot, an application that collects network-performance data useful to increase network transparency.

1.2 Which Kinds of Discrimination are Possible?

The objective of this Section is to explain which kind of discrimination technologies are available. In many cases traffic discrimination is a two-step process. In the first step the traffic is classified: a traffic flow is given a label (e.g., 'web', 'video') that depends on certain features of such flow. In the second step the traffic is handled by the network according to its label. We will start by reviewing available traffic-classification techniques, then we will discuss how, once the traffic is classified, it can be handled differently.

1.2.1 Traffic Classification

The objective of traffic classification is to assign a traffic flow to a class, selected among N classes decided in advance (e.g., 'web', 'video', 'other'). For example, the traffic flowing from 127.0.0.1:54321/tcp to 127.0.0.1:80/tcp may be assigned to the 'web' traffic class (because port 80/tcp is typically used by web traffic). Traffic classification provides information on the protocols (or class of protocols) that are active in the network at any given time. The information provided by traffic classification may be used in many ways, including to prevent attacks, to block spam and botnets, to enforce policies, to better provision the network, to guarantee a certain quality of service (QoS). Later on we will focus on policy enforcement and on QoS.

Most traffic classification techniques need to inspect the packet headers, and some of them also need to inspect the packet payload. The simplest packet classifiers only use the IP addresses and the ports. Because many applications use 'well known' destination ports [66] (e.g., port 80/tcp for HTTP, the Hyper-Text Transfer Protocol [211]) it is possible to identify the traffic class by looking at the destination port. This mechanism is, of course, very fast and is the one implemented by default by many well-known packet filters, e.g., Linux netfilter [99] and OpenBSD PF [124]. The problem of port-based classification is that circumvention is easy: an application that wants its traffic classified using another class only needs to change its destination port [232]. Such change may not be feasible for protocols that use well known ports (e.g., HTTP), but it is feasible for peer-to-peer applications in which a peer dynamically discovers the endpoint (i.e., the address, port, transport-protocol triple) of other peers. For example, BitTorrent [27] typically uses ports

6881–6889 [169], but may be instructed to use other ports.

Because applications may use random ports, it is more effective to inspect not only the ports, but also the payload. Such traffic classification technique is the so-called deep packet inspection (DPI). As described on the work of Risso, et al., [270], depending on the classification requirements, DPI may range from simple pattern matching to flow reconstruction and protocol validation. When pattern matching is used, the packet payload is matched with well-known signatures that uniquely identify protocols. For example, the BitTorrent peer protocol is known to send the byte '\0x13' followed by the string "BitTorrent protocol" within the first few TCP packets. When flow reconstruction and protocol validation are used, the DPI approach also checks whether the exchanged messages are not only syntactically but also semantically correlated. For example, a HTTP GET request in one direction should be followed by a HTTP/1.x response flowing in the opposite direction.

The main technical problem of DPI is that processing the content of packets is computationally heavy and may consume too much memory. Therefore, to make traffic classification more scalable, there is a strand of research that aims to develop more efficient classification techniques. One such technique, called service-based traffic classification [174], rests on the assumption that an endpoint is only used to provide a given service. Therefore, once an endpoint is classified (using, e.g., DPI) with sufficient accuracy, all the flows that use such endpoint are automatically classified. This technique is more scalable because one can avoid to classify the flows that use an endpoint that has already been classified.

To evade DPI techniques, one should use strong encryption: if the payload cannot be read, no pattern can be matched. Skype, for example, cannot be classified using DPI because its flows are encrypted and because its protocol is secret. However, in 2007 researchers managed to identify Skype traffic using a Bayesian classifier that checks flows characteristics such as the distribution of the inter-packet gap and the distribution of the packet length [186]. In the same vein, other peer-to-peer applications could be identified, for example, by looking at the 'social' behavior of the hosts: there are graph-based techniques that allow to detect botnets (which are a kind of peer-to-peer applications) "by targeting topological properties of their interconnectivity graph" [275]. There are also techniques that allow to classify applications by only counting the received packets and bytes, using as input the NetFlow records kept by edge and core routers [273].

To conclude: port-based traffic classification allows ISP to get a first-cut traffic classification. Such classification can be further refined, using DPI when the traffic is not encrypted and using behavioral and statistical techniques when the traffic is encrypted.

1.2.2 Traffic Blocking

The simplest technique to block the traffic is to drop undesired packets. One can implement this technique very easily using, e.g., the Linux netfilter [99] packet filter. It suffices to indicate the characteristics that the packet must satisfy. For example, one can write:

```
/usr/sbin/iptables -A FORWARD -d 8.8.4.4 -j DROP
```

to drop all the packets directed to 8.8.4.4 (the secondary Google Public DNS [53] address). This technique to block traffic can be implemented, e.g., when a certain host/port must not be reachable within a network. To circumvent this kind of blocking one needs to use a proxy server [163], a Virtual private network (VPN) [166], or Tor [247].

A more sophisticated blocking technique is the TCP reset attack, described, e.g., in the work by Weaver, et al., [302]. The network is configured to route the traffic through a middlebox that, using DPI, searches the traffic for certain patterns. When there is a pattern match, the middlebox sends a RST segment to one or both endpoints. Such RST segment is processed by the TCP/IP stack, which immediately aborts the TCP connection. This technique is more flexible than filtering a host/port, because it allows one to censor only specific pieces of content (e.g., the ISP allows users to reach a certain website, but severs the connections that request censored URLs). On the other side, however, this technique is of course less efficient than filtering a host/port. As for host/port blocking, a proxy server [163], a Virtual private network (VPN) [166], or Tor [247] can be successfully used to circumvent this kind of attack.

Another lightweight technique that can be used to prevent people from access a host is DNS filtering. The DNS server, in fact, can be configured to return an error (or a fake entry) when a certain domain name or class of domain names are requested [45]. To circumvent this kind of blocking, one typically only needs to use an alternative DNS server: for this reason this technique is lightweight (compared to the others).

Whatever the technique used, we can conclude that the net effect of traffic blocking is that an application does not work as expected and returns an error.

1.2.3 Traffic Throttling

A technique to combine traffic classification and traffic throttling is the following: packets are classified when they enter the ISP network and marked accordingly; then, depending on how the packet is marked, a diverse service is provided by routers within the ISP network. The Differentiated services (DiffServ) framework, for example, defines a number of broad classes, including: the default best-effort class, and the expedited-forwarding class, which is "dedicated to low-loss, low-latency traffic" [151]. DiffServ is typically implemented using router-queue management algorithms such as Class Based Queuing (CBQ), Priority Queuing (PRIQ) and weighted random-early-detection (WRED) [125] [31].

CBQ is a technique that allows one to divide a queue (henceforth called 'parent queue') in sub-queues: a bandwidth is assigned to the parent queue and portions of such bandwidth are assigned to each sub-queue. Moreover, it is possible for a queue to borrow bandwidth from its parent queue, when there is unused bandwidth. Assuming that there are two classes only (best effort and expedited forwarding), CBQ can be used to assign, say, 1/3 of the bandwidth to the best-effort queue, as well as to allow such queue to borrow from its parent queue. In such scenario, the best-effort class may use up to parent-queue bandwidth when the load is low (e.g., during the night) but is forced (by network losses) to yield when more and more expedited-forwarding traffic is injected into the network. In another, related scenario, we can assume three classes: peer-to-peer, best effort and expedited forwarding.

In such second scenario, the peer-to-peer class may be assigned 1/3 of the bandwidth with no permission to borrow bandwidth from the parent queue, thereby reducing the impact of the peer-to-peer traffic on the other best-effort traffic.

PRIQ is a technique that allows one to divide a router queue (henceforth 'the parent queue') in sub-queues. Each sub-queue is then given a priority. Queues with equal priority are processed in a round-robin fashion. Queues with low priority are always processed after queues with high priority. This means that high-priority traffic may starve the low priority queues. Assuming two traffic classes (best effort and expedited forwarding) and assuming that the best-effort class is given low priority, the best-effort class may use up to the parent-queue bandwidth when the load is low, but it may be close to impossible to transmit best-effort traffic when the high-priority traffic grows.

WRED is based on the random-early-detection (RED) algorithm. In RED the packet drop probability: is zero when the queue length is lower than a minimum threshold, increases linearly when the queue length is greater than the minimum threshold and lower than a maximum threshold, is one when the queue length is greater than the maximum threshold. The difference between RED and WRED is that on WRED each traffic class has its own thresholds, which are set to provide better quality of service (QoS) to the high-priority classes. Assuming two traffic classes (best effort and expedited forwarding) and assuming that the best-effort class is given low priority, when the queue starts to build up, the best-effort class will experience higher packet-drop probabilities than the expedited-forward class.

To conclude, whatever the technique used to manage the queues, low-priority traffic is expected to receive a worse service from the network when the network is loaded (i.e., during peak hours).

1.2.4 Traffic Routing

As regards traffic routing, there are two ways in which an ISP can use routing to discriminate traffic. The first technique is MPLS, the Multiprotocol Label Switching [158]. MPLS is often used inside the backbone of a ISP to route the traffic using a label-switching approach rather than routing tables. In practice, upon entering the MPLS cloud (which is contained within the ISP network), packets are assigned a label. Such label is then used within the MPLS cloud to route packets through virtual circuits that start at the MPLS ingress router and terminate at the corresponding MPLS egress router. Once packets reach the egress router, the ordinary routing-table-based routing resumes. Clearly, MPLS may be used to discriminate traffic: packets belonging to different classes may be routed through different MPLS circuits. For example, the MPLS circuit assigned to low-priority traffic may be longer and more congested than the one assigned to high-priority traffic.

A second discrimination technique based on routing is related, instead, with intra-domain routing. By denying to network A the permission to do peering [161] (i.e., to exchange traffic for free on an equal footing), network B may force network A to interconnect with it using a longer and/or more congested channel that involves a third network C. Of course, passing for a longer and/or more congested channel reduces the quality of service

experienced by packets, leading to worse performance (compared to the performance that one could get by interconnecting directly networks A and B).

1.2.5 Concluding Remarks

To classify traffic, ISPs can choose between a number of techniques ranging from simple port-based classification to advanced techniques that exploit the behavior of applications and protocols. Traffic that belongs to low priority classes may be: blocked, throttled exploiting router queues management, or diverted to longer/more congested channels. As a final remark, traffic discrimination may be permanent or temporary, depending on ISP policies.

1.3 Network Neutrality in Practice

In this Section we examine network neutrality in practice. One relevant aspect concerns the traffic-management provisions of mobile-broadband contracts and plans. Let us examine, for example, Vodafone UK, which (as of February, 2014) has two classes of mobile plans [145]: one for mobile phones, the other for mobile broadband (i.e., dongles, tablets, laptops and netbooks). The document describing the traffic management practices on the mobile-phone plans class [144] clearly indicates that Voice-over-IP (VoIP) technologies are not bundled with the default Internet offer: if one wants to Skype with friends, in other words, he/she must purchase VoIP as an option. (Conversely, Vodafone UK allows VoIP on the mobile plans that, instead, target dongles, tablets, laptops and netbooks [143].)

Another relevant aspect is the blocking and/or throttling of peer-to-peer traffic. In this regard, a well-known example is the blocking of BitTorrent by Comcast in 2007–2010. This example is very well recounted, e.g., by the Electronic Frontier Foundation (EFF) blog post "FCC rules against Comcast for BitTorrent Blocking" [301]:

> A Comcast subscriber named Robb Topolski ran a tool called a packet sniffer while attempting to 'seed' (i.e., offer to others for download) files on BitTorrent and discovered unexpected TCP RST packets that were causing inbound connections to his computer to die. Based on his observations, he speculated that Comcast may have been responsible for this interference.

Subsequent studies, including one carried out by the EFF and described in the above mentioned blog post, contributed to further clarify the extent of the traffic discrimination. In particular, the excellent study by Dischinger, et al., [204] shed light on the exact discrimination methodology:

> [T]he uploads of a file were blocked only when the Comcast host has finished downloading the file and was uploading it altruistically [...] uploads were not blocked when the Comcast host was still missing some of the pieces of the file and thus, appeared to be interested in downloading [...] we conclude that the

> middleboxes which tear down BitTorrent connections maintain some per-flow state and inspect the packet payload for specific protocol messages.

According to a blog post by Wired ("Comcast No Longer Choking File Sharers' Connections, Study Says" [236]), the blocking of BitTorrent ceased by May, 2010.

A third relevant aspect concerns peering disputes between Internet companies and ISPs. In this regard, a classical example is what Ars Technica called "the Comcast/Level 3 grudgematch" [9]: a peering-related dispute that dates back to 2010 and that was apparently resolved in the summer of 2013 [208], even though few details were released.

Before that dispute, Comcast was a Level 3 customer, because it used the Level-3 backbone for transit. In other words, Comcast paid Level 3 for transporting its traffic to networks that were not directly connected to the Comcast network itself. To this end, Comcast and Level-3 interconnected in many locations within the US. At the same time, however, Level 3 also operated a Content Delivery Network (CDN), i.e., a network that hosts content such as video and that brings such content close-enough to customers so that it can be downloaded at high speed, high quality. As regards the Level-3 CDN operations, Comcast and Level 3 had a peering agreement [161].

However, in the fall of 2010 Level 3 and Netflix signed a multi-year deal for streaming services [10]. As a consequence, Level 3 wanted to deliver more traffic through the Comcast network; however, Comcast asked Level 3 to pay for delivering such additional traffic: the peering – Comcast argued – was no longer balanced, therefore, the Level-3 CDN had now become a Comcast customer. Not wanting to pay Comcast, Level 3 had the option to reach Comcast customers via the Tata transit network; however, as demonstrated by a blog post by Adam Rothschild [274], the problem was that the interconnection link between Comcast and Tata was run at capacity. Therefore, passing through such link would have implied a reduction of the quality of service, high latency and losses.

The Comcast-Tata interconnection link run at capacity is key to understand the network-performance-related implications of routing and peering-agreements between two Internet giants. Moreover, the whole Comcast-Level 3 story is also key to understand the market side of the network neutrality battle between such giants [188]: on the one hand there are ISPs, such as Comcast, that control the eyeballs to which the content providers, such as Netflix, want to serve content; on the other hand there are the content providers that have content to which users want to have access; in the middle there are organizations, such as CDNs and Internet exchange points (IXPs), whose job is to interconnect ISPs and content providers. In the following we will not debate the Internet market structure, which is well-beyond the scope of this thesis, however, we wanted to mention the critical point of interconnection agreements because it is one of the current hot topics of the network neutrality debate.

1.4 A Tour of Network Neutrality Solutions

In this Section we describe the different ways in which network neutrality issues are (or could be) tackled. The first option is to enshrine the network neutrality principle into the

Law, defining what ISPs can and cannot do. The second option is the disclosure of user data by content providers, which, if the usage patterns of many users are combined, allows to get a picture of the quality experienced by a given Internet service (e.g., streaming). The third option are measurements from the edges, which allow to understand the quality that broadband connections deliver to different protocols.

Let us start with a brief review of the available network neutrality laws worldwide according to the excellent Wikipedia entry about network neutrality [159]. Chile was the first country in the world to adopt a network neutrality law. According to Wikipedia, such law "forbid[s] ISPs from arbitrarily blocking, interfering with, discriminating, hindering or restricting an Internet user's right to use, send, receive or offer any legal content, application, service or any other type of legal activity or use through the Internet. To that effect ISPs must offer Internet access in which content is not arbitrarily treated differently based on its source or ownership" [159]. There is also a network neutrality law in the Netherlands, since May 8, 2012. As regards traffic management, in particular, such law states that ISPs are allowed to throttle the user traffic to "minimize the effects of congestion, whereby equal types of traffic should be treated equally", to "preserve the integrity and security of the network", [...] and to "give effect to a legislative provision of court order" (citations taken from the unofficial translation of the law by Bits of Freedom [298]). The Dutch law also contains other provisions, including one against wiretapping that forbids ISPs from using DPI to track customers behavior. There is also a network neutrality law in Slovenia, in effect since January 1, 2013; such law forces the ISPs to respect the network neutrality, which is defined as "a principle that every Internet traffic on a public communication network is dealt with equally, independent of content, applications, services, devices, source and destination of the communication" [17]. Also Israel has a network neutrality law, which was approved on February 10, 2014. According to the Israel Technology Law Blow, however, it is "important to note that the law is vague on a number of important questions – such as data caps, tiered pricing, paid prioritization, and paid peering" [68].

Let us now discuss the disclosure of user data by content providers. We consider, in particular, the 'smart disclosure' of user data which 'The GovLab Wiki' defines as "the timely release of complex information and data in standardized, machine-readable formats in ways to enable consumers to make better-informed decisions" [54]. To the best of our knowledge, no content provider provides its users with such kind of information. It is possible, however, to imagine that, if content providers (e.g., YouTube, Netflix) disclosed to users all the information related to their service (e.g., video streaming), there could be third-party services that join user-provided data with data collected by measurement tools, to provide users with a dashboard of the quality of the available broadband offers.

Let us now discuss of how network-performance measurements from the edges can help to study the network neutrality. One can measure the network performance either by running active experiments or by studying the user traffic. The performance measurements, published openly, increase the network transparency, because they expose the general behavior of the network. With more data comes the possibility to run statistically relevant studies. In particular, if the type of discrimination is known (e.g., blocking, throttling), one can analyze the data to search for evidence of such discrimination. Otherwise, if the

kind of discrimination is not known, one can use data mining techniques to discover kind of discrimination of which he/she was not aware.

Of course, the three options presented above are complementary. The law can define what is and what is not acceptable, however, as binding as it may be, it is toothless without proper measurement tools that allow to verify the compliance with what was enshrined by the law. Similarly, the smart disclosure of content providers data, as informative as it may be, only provides partial information: in fact, it does not cover the interconnection between two users and it does not cover protocols discrimination.

To show how the three solutions above can be complementary, let us introduce an example based on the work of Sluijs, et al., [284], which emphasizes the role of transparency[1]. In such work the authors "evaluate the effects of increased transparency about the actual quality of broadband Internet [...] and compare four treatments in which end-users have different amounts of information about broadband quality" to conclude that:

> (1) more information about the quality of a broadband connection leads to higher total surplus and higher end-user surplus;
>
> (2) quality provided by ISPs increases with the level of transparency;
>
> (3) quality and efficiency are marginally higher when full information about quality is only available to some end-users, than when all end-users have imperfect information about quality.

Here is the example: in the above framework, the quality provided by ISPs can be increased with higher level of transparency. This result can be obtained by using the law to impose to the ISPs strict quality metrics and, to verify the ISPs compliance, by deploying enough measurement tools that measure such quality metrics[2], thereby providing imperfect information to all users. In addition, certain users may be given by content providers (e.g., video streaming providers) extra information on their performance as measured from the point of view of the content providers themselves. This disclosure would allow such users to have perfect information on their broadband networks, since they have measurements from both the regulator point of view (compliance with the imposed metrics) and the usage point of view (content providers data). Thus, according to the framework defined by Slujs, et al., [284] there will be higher quality and efficiency.

Having discussed all the available options and how they can be complementary, we now focus on measurement tools. In particular, we will concentrate on tools that perform measurements from the edges designed to increase network transparency, thereby helping to study network neutrality.

---

[1] Another interesting work on the key role of transparency in the Internet Service Providers industry is "Transparency and Broadband Internet Service Providers" by Faulhaber [209].

[2] Such quality metrics may be defined as sort of "nutrition labels" for the Internet, as proposed by Sundaresan, et al., in "Helping Users Shop for ISPs with Internet Nutrition Labels" [288]

1.5 Structure of the Thesis

In this Section we connect the network-neutrality discussion held so far with the main scientific contributions provided by this thesis. To do so, we will first describe the content of each Chapter, and then we will summarize the most relevant contributions.

We started this doctorate to design a software platform able to run network-neutrality measurements from the edges. At the time (as we will see in Chapter 2) most tools were interactive tools that performed only a test able to identify a specific kind of network-neutrality violation. We aimed, instead, to design and build a more general tool able to host many measurement modules, able to adapt to new kind of discrimination by downloading new/updated measurement modules. This tool was to be called Neubot, the network-neutrality bot.

We designed Neubot with network performance in mind because network neutrality, network transparency and network performance are tightly related, as we have previously seen in this Chapter. We describe the Neubot architecture in Chapter 3. Neubot is a tool that runs in the background and periodically performs active network-performance tests with either test servers (client-server mode) or other Neubots (peer-to-peer mode). Neubot tests, which emulate several application protocols, are implemented as plugins that can be updated independently of the Neubot core. Neubot can adapt the testing policy (i.e., which test to run and with which server/peer) depending on the ISP and on the geographic location of the Neubot instance. Neubot can run arbitrarily-long measurements because it monitors the hosting computer load to understand when it needs to release system and network resources.

We discovered that Neubot was suitable to run, from the edges, not only network-neutrality-related experiments but, more generally, many kind of network-performance-related experiments. Although the most advanced features (peer-to-peer tests, arbitrarily long tests, automatic adaptation of the testing policy depending on the ISP/location) were not yet implemented, in fact, Neubot was already flexible enough to allow us to run the large scale network-experiments campaigns described in Chapters 6–9.

While such experiments campaigns were in progress, Neubot was added to Measurement Lab (M-Lab), a consortium that provides open measurement tools with servers and other services useful to run active network measurements. Later, we won an unrestricted $52,500 research grant awarded by Google (a member of the M-Lab consortium) to continue Neubot-related research. We recount these two relevant events in the Neubot history, as well as other events, in Chapter 5. In the same Chapter we also provide up-to-date Neubot numbers (e.g., the number of performed tests/day).

Returning to our measurements campaigns, in Chapter 6 we describe the 'HTTP Speedtest' experiment. We ran this experiment to measure the broadband speed of the users running Neubot. As we learned during this experiment campaign, however, there is more in broadband speed measurements than the download/upload speed alone. Further studies and a very fruitful interaction with Matt Mathis (a well-known researcher in the network-performance field and a member of M-Lab) convinced us to change the design and the objectives of the HTTP Speedtest test, to produce results more correlated with

network quality than with the raw broadband speed.

Following up the HTTP-Speedtest changes to increase its sensitivity to network quality, we became interested in the link between the performance measured at the application level and key network-level parameters. We studied, in particular, the link between the goodput (i.e., the application-level speed) and the packet loss rate (PLR). As recounted in Chapter 7, we designed and studied two new models to estimate the PLR from application-level measurements. We validated such models with many experiments, including a large-scale measurement campaign that was only possible because the Neubot platform allowed us to deploy a new experiment onto the 1,000+ Neubot instances installed worldwide. The two models that we studied are relevant to network-performance (and to network-neutrality) because one of the ways in which an ISP can discriminate packets, as we have seen in this Chapter, is by increasing the PLR experienced by a network flow (e.g., using the weighted-random-early-detection router queuing discipline).

In addition, we also used Neubot to study the performance of the Dynamic-Adaptive-Streaming-over-HTTP (DASH) streaming technology, as we will see in Chapter 8. Motivated by the desire to allow the research community working on the DASH to run simulations involving diverse DASH rate-adaptation algorithms, this network experiment is also interesting from the network-neutrality point of view, because video streaming is currently one the most widely-used Internet applications. As for the 'raw' test, this experiment was only possible because more than 1,000+ users already installed Neubot and because Neubot allowed us to easily deploy a new experiment on such Neubot instances.

Starting from the Summer of 2011, Neubot also hosted a BitTorrent test that emulates the BitTorrent protocol, as we will see in Chapter 9. We run this experiment to measure the network performance that can be achieved using the BitTorrent protocol. The BitTorrent test methodology is similar to the HTTP-Speedtest methodology, therefore, the results of the two tests can be compared. In fact, as we will see in the last two Chapters, we used Neuviz (the Neubot visualizer, described below) to study Neubot data and, in particular, to compare the performance of the BitTorrent test to the one of the HTTP Speedtest.

In addition to the four measurements campaign described above, we also studied enhancements of the Neubot architecture. In particular, from the Summer of 2014 onwards, we worked to automatically adapt the Neubot testing policy. To this end, we developed Neuviz (the Neubot visualizer), described in Chapter 10. Neuviz is a software tool that imports and processes the Neubot data. The Neuviz web interface, in turn, allows to navigate the Neubot data, showing statistics on the number of tests and users per country as well as on the median speeds measured by the Neubot tests. After further developments, we expect Neuviz to automatically provide the statistics needed to tailor the test policy to the ISP and geographic location of a Neubot instance. For example, we can request more frequent tests to the Neubot instances deployed inside the network of ISPs for which we have few tests; we can request in-depth tests inside the network of ISPs for which the median speed of a certain protocol is below the median measured for other protocols.

To validate Neuviz, as reported in Chapter 11, we loaded into Neuviz the results collected during one-and-a-half years by the Neubot HTTP Speedtest and BitTorrent tests.

Despite Neuviz being only a prototype, we were able to find three specific anomalies concerning the distribution of the median values measured by HTTP Speedtest and BitTorrent in different countries. Such anomalies are now to be investigated (so far, manually) with more specific tests to understand whether they are caused by our testing methodology or by the behavior of the network.

To conclude, the major contributions of this doctoral thesis are: the description of the design, implementation and validation of the Neubot architecture (which has been proved suitable to run diverse kind of network-performance measurements from the edges) and the two models allowing to correlate application-level measurements with the network-level packet-loss rate. The models, in particular, are – in perspective – one of the required building blocks to allow the class of applications like Neubot to implement distributed network-neutrality measurements. By linking a certain application-level measurement with its probable network-level cause, in fact, they contribute not only to *document* the behavior of the network (e.g., protocol X seems to be discriminated), but also to *explain* what happens (e.g., protocol X consistently experiences a high packet-loss rate).

Chapter 2

Related Work

In this Chapter we list projects and papers related to Neubot (see Chapters 3–9). We describe Measurement Lab first, because it is referenced by the description of many other tools, then we proceed in alphabetic order, citing the works studied more closely during the doctorate (i.e., this review of the related work does not aim to be exhaustive).

We will also discuss data visualizations and analyses performed on the data collected by the described tools. This discussion is relevant to Neuviz (the Neubot visualizer, see Chapters 10–11), a tool to visualize Neubot data and to facilitate Neubot data analysis. We discuss Neuviz-related work in this Chapter, rather than when Neuviz is presented, because Neuviz is a topic that spans more than one Chapter of this thesis.

Measurement Lab Measurement Lab (M-Lab) is a consortium – founded by the New America Foundation's Open Technology Institute, the PlanetLab Consortium, Google, and academic researchers – that provides a distributed server platform, currently counting more than 100 servers distributed worldwide, on which researchers can deploy the server component of their active network-performance measurement tools [206]. Neubot is one of the tools hosted by M-Lab since version Neubot 0.4.6 (released on January 24, 2012).

The problem that M-Lab tries to solve is the following: active measurement tools that aim to accurately estimate the speed of users' access links need to perform the measurements with a nearby server, while tools that aim at characterizing many network paths need to run tests with a number of servers deployed worldwide. In addition to servers, M-Lab also provides the hosted tools with a number of services, including the automatic collection and publishing of the experiments results and Web100 support (Web100 is a modified Linux kernel, described below, that allows to gather TCP-level statistics).

To be hosted by M-Lab, a tool must pass a technical and policy evaluation. At the minimum, a tool is required to be open-source and to release the collected data in the public domain. M-Lab, in fact, is committed to increase the network transparency by facilitating the job of tools that perform network-performance measurements. The data collected by M-Lab, in particular, is published on the Google-Cloud-Storage platform [34]. Moreover, the results of all the experiments are being added to BigQuery, a RESTful service to query big datasets using an SQL-like language [26].

(Beverly, et al., 2007) The paper of Beverly, et al., aims to build a map of port blocking policies, with emphasis on VPN, email and file sharing ports [184].

BISmark BISmark is an home gateway, based on OpenWrt [123], specially-instrumented to perform active and passive measurements [16] [287].

BTTest The article of Dischinger, et al., describes BTTest [204]: a tool to detect BitTorrent RST-injection practices, which has been used in 2008 to characterize the RST policy of many providers, including Comcast (as described in Section 1.3). The tool was a Java-applet-based test that users could run in the browser that later evolved to become Glasnost (a more advanced browser-based network measurement tool described below).

Regarding data, M-Lab only publishes BTTest/Glasnost data collected after January 28 2009 [33]. The Glasnost website [51] does not say when BTTest became Glasnost, however, it is reasonable to assume that the data published by M-Lab is Glasnost data only. In any case, the Glasnost website publishes three reports containing postprocessed[1] and aggregated data that most likely were collected using BTTest: the July 25, 2008 [2], the November 9, 2008 [3] and the February 1, 2009 [6] reports.

Dasu Dasu is a plugin for the Vuze BitTorrent client that combines passive and non-intrusive active measurements to monitor the ISPs' level of service [185] [278]. Dasu is open source. According to the DASU webpage [43]: "This software is released under the GPL. The source code can be found in the jar." The jar (Java compressed archive) can be downloaded from the Vuze website [146]. The Dasu webpage [43] does not indicate any open repository where the Dasu development takes place. Also, the webpage does not provide any pointer to download the data collected by Dasu.

DiffProbe DiffProbe is an active network-measurement method able to detect whether an ISP is performing delay and/or loss discrimination by comparing the delays and losses experienced by two flows: an application flow and a probing flow [230].

DiffProbe has close bonds with ShaperProbe (see below). In particular, the DiffProbe webpage at NetInfer.com [44] states: "We have released a module of DiffProbe called ShaperProbe which aims to detect if the ISP is doing traffic shaping." In fact, many ShaperProbe source files [135] begin with this comment: "This file is part of Diffprobe."

Unlike ShaperProbe, the source code of DiffProbe was never released. The DiffProbe webpage [44], in fact, still says: "The tool is in development. If you are interested in using DiffProbe, please fill the form below, and we will contact you once we release the tool."

The DiffProbe webpage does not provide any pointers to publicly available data.

[1] The original traces are not available, but there is a table that indicates, for each country, the total number of tests and the number of tests in which BitTorrent was blocked.

Fathom Fathom is a Firefox extension that implements a browser-based network measurement platform [202] providing "a wide range of networking primitives directly to in-page JavaScript [...] TCP/UDP socket access, higher-level protocol APIs such as DNS, HTTP, and UPnP, and ready-made functionality such as pings and traceroutes." [47]

Glasnost Glasnost – the evolution of BTTest – is a client-server browser-based Java-applet developed by the Max Planck Institute for Software Systems [205]. It is designed to make protocol shaping and blocking practices more transparent to end-users by performing active transmission tests between the user computer and M-Lab servers.

Glasnost's test supports a wide-range of application protocols (e.g., BitTorrent, Emule, and NNTP) and is constructed to indicate whether the shaping-causing feature is the port number or the payload content. To simplify the process of generating new tests, Glasnost includes a mechanism to construct Glasnost tests from packet traces, by generating a script written in a simple domain-specific language. In fact, the Glasnost application and the Glasnost server are just interpreters of such language.

Glasnost's guiding principle is that the shaping technology must have an impact on the application-perceived performance. To quantify it the test compares the goodput of two back-to-back 60-second TCP flows that are identical except for the feature supposed to trigger discrimination (e.g., to identify BitTorrent-protocol discrimination a regular BitTorrent flow is compared with a random-payload reference flow). This is repeated five times to estimate cross-traffic noise and single out noisy paths. For paths that are flagged as non-noisy, Glasnost computes the difference between the two flows' maximum goodput and reports shaping when the difference is greater than 20%.

Results of Glasnost's tests are automatically published on the Google Cloud Storage [33], under a Creative Commons Zero waiver [30], by M-Lab. Glasnost's repository can be easily cloned from Google Code [50]. The Glasnost source code is released under the terms and conditions of the three-clause BSD license.

As regards visualizations of Glasnost data, Michael Bauer, data wrangler[2] at the Open Knowledge Foundation, created a visualization of Glasnost data, which shows on the world map the percentage of tests that Glasnost detected as shaped [20]. The user can filter the dataset to show only the results related to a single protocol emulated by Glasnost.

The Glasnost data were also analyzed in the context of "The Network is Aware" research project of Dr. Milton Mueller (see below).

[2] According to Wikipedia: "Data munging or data wrangling is loosely the process of manually converting or mapping data from one "raw" form into another format that allows for more convenient consumption of the data with the help of semi-automated tools. This may include further munging, data visualization, data aggregation, training a statistical model, as well as many other potential uses. [...] A data wrangler is the person performing the wrangling. In the scientific research context, the term often refers to a person responsible for gathering and organizing disparate data sets collected by many different investigators, often as part of a field campaign." [150].

Grenouille Grenouille [56] is a French[3] nonprofit association whose subject is the "promotion of the broadband Internet" [57]. In particular, Grenouille (citing Wikipedia [60]) "publishes *la météo du net*, an observatory of real time performance of French broadband service providers." Grenouille develops a number of clients and a server. All the software is open-source, and the Git repository used by Grenouille is browsable online [59]. The following considerations apply to the Grenouille Python client [129], chosen among the available Grenouille clients because such client implements more features than the other Grenouille clients written in programming languages known to us.

Grenouille (including in the term both the Python clients and the Grenouille servers) has an architecture similar to Neubot, but a slightly different goal: to measure ISP backbone congestion. Periodically, 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, ping tests, and traceroute measurements are not (much) biased by other ISPs, and hence it is possible to evaluate the average quality of service. The Python client has also an invalidation mechanism by which tests are aborted when the Python client detects that, during the test, the user heavily used the network. Tests results are reported to the Grenouille servers, which process the results, and produce statistics.

The results are published on the Grenouille association website [56], both in a dashboard that highlights the most recent measurements and in charts (monthly charts, daily charts, per-city charts). However, we have found no links to download the raw data collected. In this regard the FAQ [58] states that aggregate daily data is kept on the Grenouille servers for some weeks and monthly data is kept on the servers for longer time[4]. Also, the FAQ states that Grenouille will not share the collected data with anyone and specifies that the French law does not allow it to disclose the collected data with third parties[5].

HoBBIT Developed by the University of Napoli networks and multimedia research group, HoBBIT (Host Based Broadband Internet Telemetry), is a platform for scheduling and running broadband performance measurements [63]. The source code of HoBBIT is "available on request." Data are "gathered in anonymous way [. . .], stored at University of Napoli and will be used exclusively for research purposes." Fine grained results "are available to respective users." The HoBBIT website includes a nice zoomable visualization that allows one to see the results collected by HoBBIT in Italy. The visualization is zoomable and the aggregation level ranges from the Provinces to the zipCode.

The HoBBIT architecture includes clients, test servers and control servers. Every five

[3]Most online material concerning Grenouille is in French; since we do not master the French language, we used Google Chrome's translator to translate such material to English.

[4]The original FAQ entry reads: "*Les données des courbes journalières assez peu de temps... de l'ordre de quelques semaines... c'est pas assez, nous en sommes conscients et nous travaillons à allonger cette durée. Les données des courbes mensuelles sont conservées beaucoup plus longtemps.*"

[5]The original FAQ entry reads: "*Remarque : Notre fichier est inscrit á la CNIL et le fait de divulguer des informations sans l'accord des intéressés est condamné par la loi française.*"

minutes the clients connect to the HoBBIT servers to download a list of experiments, then the clients sequentially execute the experiments and report the results to the servers. The HoBBIT measurement framework is only loosely coupled with the measurement plugins, with the advantage that already existing tools can be used and the disadvantage that scripts (in bash and awk) are needed to glue the existing tools with the central framework. Currently, HoBBIT is running an experiment called Basic Performance Evaluation, in which basic performance metrics are evaluated. HoBBIT measures round-trip time, latency, jitter, packet-loss rate, upload and download throughput using TCP and UDP.

The paper by de Donato, et al., [200] describes HoBBIT and presents an interesting analysis of the performance measured using HoBBIT in Italy.

MeasrDroid MeasrDroid [81] is an Android application, developed by the Technical University of Munich, that measures a variety of metrics, ranging from hardware information to active probes (e.g., DNS queries, traceroute).

MisuraInternet.it MisuraInternet.it [84] is the Italian Government initiative to measure the broadband performance in Italy. It has two objectives: to perform authoritative measurements of the quality of the most popular Italian ADSL offers[6]; to allow citizens to perform their own measurements to evaluate the ISPs' quality, by running the Ne.Me.Sys and 'MisuraInternet Speed Test' pieces of software[7].

MobiPerf MobiPerf [88] is an Android application for performing network performance measurements (traceroute, HTTP, TCP speed test, ping, DNS lookups). It is developed by the University of Michigan, the University of Washington and M-Lab. It runs periodic measurements in the background, and it can also run on demand measurements.

mPlane mPlane [297] is an ongoing EU-funded FP7 ICT research project lead by the Dept. of Electronics and Telecommunications of the Politecnico di Torino that aims at building an intelligent ISP-oriented measurement plane to spot application and performance problems. Based on passive and active measurements from inside the network as well as from user devices, mPlane seeks to identify the most-commonly-used applications (via DPI and data-mining-based traffic classification), characterize users' connections (e.g., in terms of capacity, latency, and goodput), and perform Quality of Experience estimations (e.g., by emulating YouTube's video player). Collected data include business- and privacy-sensitive information and is not meant to be shared with the public at large. An intelligent reasoner will be developed in the context of the mPlane project to drill down the cause of performance problems and raise alarms.

[6]In Italian: "*effettuare misure certificate al fine di comparare la qualitá delle prestazioni offerte da ogni operatore, relativamente ai profili/piani tariffari ADSL piú venduti.*"

[7]In Italian: "*mettere in condizione l'utente/consumatore [...] di valutare autonomamente la qualitá del proprio accesso ad Internet [...].*"

NANO NANO (Network Access Neutrality Observatory) [292] [293] is an agent that passively collects performance data from users' computers and uploads them to the NANO servers. More specifically, the passive information collected by NANO encompasses "the number of packets transferred for each active flow per unit time, as well as tracks for unexpected events like packet loss, and TCP RST packets [...] the application that owns the flow [...] the load on the client computer" [96].

The NANO servers use statistical *stratification* techniques that group clients in clusters where the performance difference depends on the ISP's policy and not on *confounding factors*, i.e., "variables correlated with both the treatment variable (the ISP) and the outcome (the performance)" [293]. The NANO authors identified three categories of confounding factors: client-based confounders (e.g., the operating system, the browser), network-based confounders (e.g., the location of the servers used, hence the latency) and time-based confounders (e.g., weekly and daily traffic patterns).

As of this writing, NANO is a discontinued software project. Its sources, released under the terms and conditions of the GNU General Public License version 3, can be downloaded from the website of the Georgia Tech Network Operations and Internet Security Group (GtNoise) [95]. The data collected by NANO is not publicly available.

NDT The Network Diagnostic Tool (NDT) is a network-measurement Java applet that measures the download and upload speed between the user computer and an M-Lab server [192]. During the measurement, the server uses the modified Web100 Linux TCP/IP stack (see below) to expose the state variables of TCP during the transfer. In addition to the Java applet an NDT command-line application is also available. The NDT test methodology is also integrated by uTorrent clients [258], which use it to estimate the available bandwidth and tune their settings for optimal performance.

Results of NDT's tests (including the many tests run by uTorrent clients) are automatically published on the Google Cloud Storage [35], under a Creative Commons Zero waiver [30], by M-Lab. NDT sources are released under the three-clause BSD license. The NDT Subversion repository is hosted by Google Code [97].

The M-Lab team developed a visualization of NDT data that shows many indexes (e.g., the number of tests, the download and the upload speed, the round trip time) on the world map [18]. Such visualization allows one to aggregate the data by ISP and by geographical dimension (country, region/state, city), and it also allows one to compare the performance of multiple ISPs at different geographical levels.

Dominic Hamon, a software engineer at Google and at M-Lab, developed visualizations (and a video) [223] that show, on the world map, a point indicating the latitude and the longitude of each client that runs a test towards an M-Lab server, using the data collected by the NDT tool and retrieved via BigQuery.

The American researcher Collin Anderson, with funding from the Center for Global Communication Studies at the University of Pennsylvania's Annenberg School for Communication and Google Research, authored a study in which NDT data (and, in particular, the Web100 variables related to NDT tests) are used to detect the usage of throttling as a

mechanism for censorship in Iran since the 2009 elections [172].

Netalyzr Netalyzr is a Java program that "probes for a diverse set of network properties, including outbound port filtering, hidden in-network HTTP caches, DNS manipulations, NAT behavior, path MTU issues, IPv6 support, and access-modem buffer capacity." [237] It runs as a Java Applet or as a command line application. It is also available as an Android app on the Google Play store.

NetPolice NetPolice (formerly NVLens) is an active tool that combines traceroute-like characteristics with protocol-awareness to measure the packet-loss rate of portions of the backbone and spot violations of network neutrality [305] [306]. To the best of our knowledge, no data collected by NetPolice was ever publicly released. In the same vein, as far as we know, no source code was ever released.

NNMA NNSquad Network Measurement Agent (NNMA) – an initiative of the NNSquad (Network Neutrality Squad) project [117], animated by the American technologist and activist Lauren Weinstein [73] – is a prototype piece of software that detects certain spoofed RST segments [7]. The assumption on which NNMA rests is that, when a RST is injected, it is received much earlier than expected especially if the middlebox that inject the RST is close to the receiver of the RST. NNMA is now a discontinued piece of software, since its last release (corresponding to version 2.0.0.6 beta) occurred on 29 May 2008. Its sources, available under the terms and conditions of the GNU Lesser General Public License version 2.1, can be downloaded from the NNMA web page [7]. As far as we know, no data collected by NNMA was ever publicly released.

NPAD Network Path & Application Diagnostic tool (NPAD) diagnoses common problems affecting the last mile, especially problems that commonly affect TCP application [252]. To diagnose the cause of the poor performance, NPAD uses Web100 (see below) and TCP performance models (e.g., the Mathis model [250]). NPAD is hosted by M-Lab and performs its active network test with M-Lab servers. Results of NPAD's tests are automatically published on the Google Cloud Storage [37], under a Creative Commons Zero waiver [30], by M-Lab.

OONI Developed by the Tor Project, OONI, the Open Observatory of Network Interference [212], is an application that aims to detect network censorship. To this end, OONI implements a number of tests to reveal content blocking (e.g., whether a certain domain cannot be resolved from within a certain ISP) and/or traffic manipulation (e.g., whether a transparent HTTP proxy in the middle tampers with the user requests).

Tor (the piece of software that empowers the namesake anonymity network [247], developed by the Tor Project and based on the concept of onion routing[8]) is used both to run

[8] According to Wikipedia [160]: "[o]nion routing [...] is a technique for anonymous communication over a

some OONI tests and to (very anonymously) send the test results to the OONI servers.

OONI is free software, released under a two-clause BSD license, and its sources are hosted by the Tor Project's Git repository [122]. The raw results of OONI tests are publicly available on the Tor Project website [121], under the terms and conditions of the Creative Commons Attribution license.

Pathload2 Pathload2 is a tool that measures the available bandwidth of the user's broadband connection using the 'self loading periodic streams' methodology [265]. Pathload2 is hosted by M-Lab and performs its active network test with M-Lab servers.

Portolan Portolan [220] [126] is a mobile application for Android devices that implements a number of tests, including traceroute, maximum throughput estimation, and a BitTorrent test using a methodology similar to the one of Glasnost (see above).

RespectMyNet RespectMyNet is a forum animated by the French association La Quadrature du Net, the Dutch association Bits of Freedom, et al., that allows users to report (and confirm) evidence of network neutrality violations [131].

SamKnows SamKnows is a broadband measurement company. It sells "whiteboxes", hardware appliances that must be installed as a proxy between the broadband router and the other devices connected to the Internet. From such vantage point, a whitebox is able to monitor the users' Internet usage. Therefore, it can launch performance measurements only when users are not using the Internet [64].

ShaperProbe ShaperProbe is a tool (and an active probing methodology) to detect the presence of a traffic shaper deployed between two hosts and infer the shaper characteristics [231]. ShaperProbe infers the peak rate, the shaped rate and the elapsed time after which the shaper reduces the speed of a flow from the peak rate to the shaped rate.

The ShaperProbe test methodology is the following. Initially, ShaperProbe estimates the capacity, C, of the end-to-end path. Afterwards, the ShaperProbe sender starts sending at rate C and the ShaperProbe receiver takes a snapshot of the receive speed every N milliseconds, dividing the number of bytes received in the $[i \cdot N, (i+1) \cdot N]$ interval by N. The receive speed is then added to a vector containing all the speeds measured so far. Once the new entry is added to the vector, the ShaperProbe receiver uses statistical techniques on such vector, to detect whether there is a transition from the peak rate to the shaped rate. Once the transition is identified, the shaper parameters are estimated.

---

computer network. Messages are repeatedly encrypted and then sent through several network nodes called onion routers. Like someone peeling an onion, each onion router removes a layer of encryption to uncover routing instructions, and sends the message to the next router where this is repeated. This prevents these intermediary nodes from knowing the origin, destination, and contents of the message."

ShaperProbe is hosted by M-Lab and performs its active network test with M-Lab servers. Its Git repository [134] can be cloned from Google Code. The sources are released under the terms and conditions of the GNU General Public License version 3.

The data collected by ShaperProbe are automatically collected and published (under a Creative Commons Zero waiver [30]) on the Google Cloud Storage [38] by M-Lab.

SpeedTest.net The well-known SpeedTest.net web site [136] provides a network measurement, Flash-based test that relies on many parallel HTTP connections to estimate the download and upload broadband speed of the user's connection, using a methodology that is documented, e.g., in "Understanding Broadband Speed Measurements" [180]. The data collected by SpeedTest.net can be browsed online and downloaded in CSV format from the NetIndex.com website [19]. The data are available under the terms and conditions of the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International license.

Switzerland Switzerland, developed by the Electronic Frontier Foundation (EFF) international non-profit digital rights group, was a semi peer-to-peer application designed to detect the modification or injection of packets with the help of a central server [8]. To the best of our knowledge, no publicly available data was ever released concerning the results of the network-interference tests conducted using Switzerland.

The Network is Aware Dr. Milton Mueller is the principal investigator of "The Network is Aware" research project at the Syracuse University School of Information Studies [21]. The objective of the research is to investigate "whether deep packet inspection is changing the way the Internet is governed."

In this context, Glasnost data was analyzed and correlated with economic and political variables [173]. The findings of this study are the following. They found that "half of the studied ISPs are actively using DPI (deep packet inspection) in their networks, although to varying degrees"; that "bandwidth scarcity, cost of bandwidth, low levels of competition [...], high levels of government censorship and weak privacy protections are correlated with higher DPI use"; that the strength of the copyright industry and network security are not drivers of DPI. To decide whether DPI was used in a country or not, the authors processed the Glasnost results and selected the results subset in which the discriminating feature was the payload content (see above for a description of how Glasnost works).

In the context of "The Network is Aware", the following Glasnost-data visualizations were also developed [22]: an interactive table that shows which ISPs seem to shape (or block) BitTorrent; a visualization that displays the "top throttlers" ISPs from 2009 to 2012; a visualization that shows alleged BitTorrent shaping in selected countries.

(Weaver, et al., 2009) This paper describes a rich set of heuristics to classify incoming RST segments, being able not only to identify the spoofed RSTs, but also to tell, with a certain degree of confidence, which device injected the RSTs [302].

Web100 Web100 is a Linux kernel patch that instruments the TCP/IP stack to gather a wide range of TCP state variables and statistics [251]. For example, Web100 allows to read the current congestion window, to know the number of congestion signals received so far and to understand the impact of the receiver's buffer on the TCP performance. The Web100 kernel is installed on all M-Lab servers, therefore, all M-Lab tools can optionally use Web100.

WindRider WindRider was a mobile application that performed active goodput tests using different ports. It also performed passive tests such as measuring the delay to load web pages. In both cases the results were uploaded to central servers [14]. To the best of our knowledge, no publicly available data was ever released concerning the results of the tests conducted using WindRider.

Concluding Remarks Like many works described in the non-exhaustive review presented above (e.g., NDT, SpeedTest.net), Neubot works at the application level and actively measures the network performance. But, unlike many of them, Neubot runs in the background and performs periodic tests (as Grenouille does). Also, Neubot implements more than one test and has an architecture that allows us to update the available tests, which are implemented as plugins. We spent a lot of effort to design Neubot as a platform on which one can deploy new experiments, as opposed to a single-purpose tool.

Neubot and NANO have in common that they do not assume that a particular discrimination technique is in place, rather they collect data to be analyzed later. However, as regards Neubot, the process of analyzing the results to infer facts about the network neutrality has not yet started. In Chapters 10–11 (as well as in the following Section) the reader will read about our Neuviz effort that should allow us in the future to automatically spot anomalies to be investigated with more specific tests.

Neubot and OONI have in common that both are platforms on which more tests can be easily added, therefore, they have a similar architecture. However, Neubot is focused on the network performance, while OONI is focused on network censorship.

Neubot and HoBBIT have a similar, flexible architecture, but different goals. Neubot aims to measure network performance using a variety of protocols, to increase network transparency; HoBBIT focuses on broadband quality metrics.

As regards data visualization and analysis, Neuviz (see Chapters 10 and 11) is based on the world map, like NDT visualizations. However, Neuviz is optimized for complex data analysis and uses precomputed data, while the NDT visualizations are more interactive and fetch the data from BigQuery on demand.

The aim of Neuviz is similar to the aim of the Glasnost visualizations; both Neuviz and the Glasnost visualizations, in fact, intend to make access networks more transparent by, respectively, showing anomalies and alleged shaping.

Chapter 3

The Neubot Architecture

In this Chapter we describe the Neubot architecture, which is based on Neubot proper, a piece of software that can be installed on PCs and smartphones and that periodically runs active network-performance tests emulating several protocols. The Neubot tests are performed both with dedicated test servers (client-server mode) and with other instances of Neubot (peer-to-peer mode). Unlike other network-performance tools that run a single kind of test, Neubot is designed for flexibility. Neubot tests, in fact, are implemented as plugins that can be updated independently of the core Neubot application. Moreover, the Neubot servers suggest to Neubot which network test to run, as well as the test server (or the other Neubot) with which such test should be run. The returned suggestions are adapted to the ISP and the geographic location of the Neubot instance. Neubot is also designed for openness: the program is open source and the results of the network tests are open data. The combined effect of flexibility, adaptability, and openness makes Neubot a flexible platform to collect - from the network edges - network-performance data useful to study the broadband-networks performance and, more specifically, network neutrality.

This Chapter is based on the ISTAS-2008 [201] and AICA-2010 [176] Neubot papers. However, since many details have changed since such papers were published, we significantly updated this Chapter drawing more-updated content from subsequent papers, as well as from the Neubot documentation and source code. We are currently working on a journal paper, based on this Chapter, that will provide a more thorough and up-to-date description of the Neubot architecture.

Before we describe how this Chapter is organized, one more remark: not all the Neubot architecture components described in this Chapter have been implemented yet. In this Chapter, in fact, we describe the software architecture only; i.e., the asymptote to which the implementation must aim. We refer the reader to the next Chapter for a description of the current (February, 2014) Neubot implementation.

The remainder of this Chapter is organized as follows: in Section 3.1 we start with a high-level description of Neubot; in Section 3.2 we discuss the scientific and technical challenges posed by Neubot; in Section 3.3 we describe the Neubot design goals; in Section 3.4 we enter into the details of the components of the Neubot architecture; in Section 3.5 we describe the generic-network-performance-test sequence; in Section 3.6 we describe the

software architecture of the Neubot library; in Section 3.7 we draw the conclusions.

3.1 High-Level Description

In this Section we provide a high-level description of Neubot, i.e., the starting point of our discussion, to be enriched in the following Sections. To this end, we recap the basic points that we have mentioned in the introduction to this Chapter:

1. Neubot is a piece of software that runs on PCs and mobile phones;
2. the objective of Neubot is to collect network-performance data useful to study the network performance and, more specifically, network neutrality;
3. Neubot shall run multi-protocol network-performance tests with test servers (client-server mode) and other Neubots (peer-to-peer mode);
4. Neubot (we will see later why) shall only use active network measurements (i.e., it sends data emulating a specific protocol) and shall not use passive network measurements (i.e., analysis of the user-generated traffic);
5. Neubot is helped by a number of servers, which for now we will call 'the Neubot servers';
6. to allow other researchers to discuss the Neubot methodology and/or to study the Neubot data, Neubot shall be open source and the collected data shall be open data.

To the above points let us also add that:

7. examples of protocols that Neubot aims to emulate are: the Hyper-Text Transfer Protocol (HTTP) [211], the BitTorrent peer protocol [196], the Micro Transport Protocol (uTP) [259], the Real Time Protocol (RTP) [280];
8. although it is not a protocol (because it is based on HTTP), Neubot is also interested to emulate the Dynamic Adaptive Streaming over HTTP (DASH), which is a client-driver streaming technique.

Having enumerated the fundamental tenets of the Neubot design, we can now discuss which are the main scientific and technical challenges entailed by Neubot.

3.2 Scientific and Technical Challenges

In this Section we discuss the major scientific and technical challenges entailed by Neubot: (i) how to obtain reliable network-performance measurements; (ii) how to guarantee data integrity, security and anonymity, in particular during the peer-to-peer tests. We start by discussing the latter. Such discussion, in fact, allows us to explain why Neubot does not perform passive measurements. Once we have explained that, we can proceed in good order by discussing how to obtain reliable network performance measurements.

3.2.1 How to Guarantee Data Integrity, Security and Anonymity

In this Section we discuss how to guarantee data integrity, security issues, and user anonymity. We focus, in particular, on peer-to-peer network-performance tests.

As regards data integrity, we can guarantee it by using encryption to secure the communication between a 'Neubot instance' (i.e., one specific Neubot) and the Neubot servers. By encrypting the channel, we guarantee that an attacker cannot intercept and/or tamper with the exchanged messages. In addition, we shall also authenticate the Neubot servers, so that an instance can tell whether it is talking with a legitimate server. To this end, we could create a Neubot certification authority [162] that signs the certificate of the Neubot servers that we deploy. Of course, for this strategy to work, we also need to ship the public key of the certification authority with Neubot. However, we do not see an obvious way to authenticate the Neubot instance as well. As we will see in the following, this fact leads to some interesting (from an attacker's point of view) consequences.

Because we cannot authenticate a Neubot instance (more correctly, because a Trusted-Computing-like 'remote attestation' [165] of the Neubot instance is neither feasible nor desirable for ethical reasons[1]), when we receive test results from a Neubot instance, we cannot tell whether they are the results of actual network experiments or whether they are forged by a cheating Neubot instance. It is possible, however, to defend ourselves from a single cheating Neubot instance. In fact, we can exploit the fact that the measurements are always performed by a Neubot instance and a server, or by two Neubot instances. Therefore, we can proceed as follows: we can ignore the results reported by a single Neubot instance and not confirmed by a server (or by another Neubot instance). Of course, this strategy leaves us defenseless before two cheating Neubot instances that report the results of a fake peer-to-peer test. We look forward to design a more robust cheating prevention strategy when we will implement peer-to-peer network-performance tests[2].

As regards security, the Neubot instance shall always consider to be in a hostile environment during peer-to-peer tests. In fact, because we cannot authenticate instances, we do not know whether a Neubot instance is legitimate or a malicious attacker. For this reason, when we discuss the design and the architecture of Neubot, we also discuss strategies to increase the Neubot instance security.

As regards user anonymity, it is problematic to guarantee complete anonymity in all cases. In fact, as anticipated in the previous Section, Neubot aims to release its results as open data, to allow third-parties to study them. Unfortunately, to effectively study

[1] Remote attestation allows someone, say Alice, to decide whether someone else, say Bob, can exchange data with Alice depending on whether the code run by Bob has been pre-approved by Alice, as opposed to depending on the compliance with a given protocol. For example, one could be prevented from accessing the website of a bank with a standard-compliant web browser, because the bank only trusts a certain build of Microsoft Internet Explorer or Google Chrome. This is unethical because it breaks the cultural assumption that, to interconnect, one only needs to implements the required protocols.

[2] As we stated in the introduction to this Chapter, not all the features mentioned in this Chapter are already implemented. In particular, peer-to-peer tests are not implemented yet.

the data, the IP address is needed. In fact, the IP address is key to determine the user geographic location and his/her ISP, two key pieces of information to study broadband performance and network neutrality. Now, the problem is that IP addresses are considered personal data in the European Union [260] [5] [4], and for a good reason: there are some specific cases, in fact, in which the IP address is enough to identify a specific individual. For example, let us assume that J. Random Hacker is known to be the only Neubot user in town: if the attacker knows that fact, he/she can download the Neubot results and use, e.g., GeoLite [11] to determine the IP address of J. Random Hacker. Once the IP address is known, the attacker can link the online behavior of such IP address to J. Random Hacker. Of course, this was a particularly unfortunate case to explain why IP addresses are considered personal data: in reality, it is hard to track down people using Neubot data[3]. In any case, to inform his/her users and to respect the European privacy laws, Neubot shall show its users a privacy policy explaining why we need to collect and publish the IP address and asking for the user permission to save and publish it.

Speaking of privacy, let us now discuss passive network measurements, that is, measurements that capture and analyze user-generated traffic. The positive aspect of passive measurements is that they allow a researcher to measure real traffic (while active measurements only allow to study the results of traffic emulating real applications). Considering that Neubot shall publish its results as open-data, however, passive measurements risk to reveal too much about Neubot users. Therefore, in general, Neubot shall avoid passive measurements, with one exception: it is acceptable, in fact, to capture and analyze the packets that were generated by the active experiments performed by Neubot.

Having explained why passive measurements are, in general, not in the Neubot radar, let us now focus on active measurements. In particular, let us now discuss how the Neubot architecture shall be designed to perform reliable, active measurements.

3.2.2 How to Obtain Reliable Measurements

In this Section we discuss how to obtain reliable multi-protocol active network-performance measurements. The discussion, however, will not touch measurements algorithms and more specific topics already covered by Chapters 6–9. Here, instead, we will discuss which network-related aspects shall be considered when designing an architecture suitable for collecting multi-protocol network-performance measurements:

Traffic-classification mechanism The implementation of a network-performance test is influenced by the traffic-classification mechanism active in that network (if any). If deep packet inspection is used, the traffic is classified by searching the packets for known patterns. In such case, e.g., all the packets that begin with the string

[3]Moreover, let us add that there are much more efficient and effective ways to track people online. See for example the "How unique is your web browser?" paper [207], to see how it is possible to identify people using JavaScript scripts that inspect the HTTP headers and query the browser plugins.

'HTTP/1.1' may be classified as 'HTTP responses.' Therefore, a network measurement application may impersonate a protocol by emulating its syntax. That is, to emulate the BitTorrent peer protocol it suffices to send messages having the BitTorrent packet format. On the contrary, if a behavioral traffic classifier is used, the traffic is classified according to more subtle characteristics. In such case, e.g., a peer-to-peer application may be expected to exhibit a symmetrical data exchange involving many other parties, while a server may be expected to serve large quantities of data. Therefore, a network-measurement application may impersonate an application by emulating its key behavioral characteristics. For example, to successfully impersonate a peer-to-peer application, one may need to perform network-performance tests involving multiple parties. Therefore, not knowing in advance which traffic classification mechanism is used, Neubot shall be flexible and adaptable. That is, it shall be easy to change the implementation of a network test and/or to deploy a new network test (flexibility) and it shall be possible to order a Neubot instance to use a specific test methodology with a given test server or with a specific Neubot instance (adaptability).

Daily traffic patterns The time of the day in which a test is run is relevant. In fact, the load of an ISP network is not constant, rather it is highly correlated with the habits of its users. For example, many residential ISPs experience high load in the evening and experience low load during the night. Conversely, a university (or company) network typically has high load during work hours and low load in the evening and during the weekend. In both cases during the rush hours the network may be so congested that the performance is reduced. Therefore, to investigate whether the network performance is correlated with daily patterns, Neubot shall be an always-on tool that performs frequent measurements (e.g., every half an hour).

Bandwidth caps A bandwidth cap is the maximum number of bytes that a user can download (or upload) before being penalized by his/her ISP [28]. In many cases the bandwidth cap is an explicit mobile-or-broadband contract provision, but there are also cases in which the bandwidth cap is not known by the user. When the bandwidth cap is a contract provision, it is often computed over the period of a month. In such cases, the contract should also indicate what happens once the cap is reached. Typically, one of these three things happen: the access-network speed is reduced, the user is charged for the excess bandwidth, the user is cut off the network. Conversely, when the cap is not transparent, it is often computed over a short period (minutes or hours) and, when the cap is reached, the user access speed is reduced. Therefore, Neubot shall allow users having a bandwidth cap to select a lightweight testing profile. Moreover, each Neubot instance shall be identified by a unique random number. In turn, such random number allows who analyzes the data to run time series analysis on the data collected by a Neubot instance[4]. which may

[4]As regards privacy, this random unique identifier is not personal data, since it is random. It may

help to reveal hidden bandwidth caps.

Powerboost The so-called 'powerboost', often employed in cable networks, is a mechanism by which the access link speed yields high peak speeds at the beginning of a data transfer, and lower-than-peak speeds afterwards [181]. To measure the effect of the powerboost, Neubot shall run long network experiments.

User activity Speaking of long network experiments, they not only allow to expose the powerboost dynamics, but they also allow to perform more accurate network measurements. On the flip side, however, a long network test may be annoying for users: if the long test runs in the foreground, in fact, the user may not be willing to wait for a long time (as recounted by the Glasnost authors [205]), while, if the long test runs in the background, it may clash with other foreground user flows. Therefore, to perform long foreground network tests one needs to incentivize users to wait; to perform long background network flows one needs to avoid clashing with the other flows. To this end, one could monitor the computer state (network load, CPU, memory) to decide when to start and stop long background tests. Having weighted the pros and the cons of both background and foreground tests, we designed Neubot to run background tests only, because we could not easily find a mechanism to incentivize users to run long foreground tests.

User location The user location is a relevant parameter in mobile networks. For example, mobile users running network tests from a location in which the signal-to-noise ratio (SNR) is low will typically experience unsatisfactory network performance. Moreover, the user location is also relevant when one wants to analyze the digital divide. For example, users living in a hamlet may have significantly lower speeds than users living near the city center. Therefore, as anticipated above, Neubot results shall include the IP address of the Neubot user; in turn, the address allows the researcher analyzing the data to estimate (via geolocation) both the user location and the user autonomous system (AS), which is a routing domain within an ISP.

Service discrimination Service discrimination is orthogonal to protocol discrimination. For example, video traffic may be slowed down not because it is video traffic but because it comes from a well-known content provider (e.g., YouTube). Because Neubot shall only run active measurements, it is problematic to perform network measurements useful to study this kind of discrimination.

We have now concluded our discussion of the most relevant technical and scientific challenges posed by the Neubot architecture. In the following Section we will combine all what

---

help, however, to discover that a Neubot instance changed location and ISP over time. This fact is not problematic for landline Neubot users, because they do not change location, but may become problematic for mobile users: by knowing the unique random number bound to a person, one can use Neubot data to track the location of such person. Since this kind of tracking is undesirable, when we will implement Neubot for mobile, we will modify the implementation to reduce the risk of tracking.

we said so far to identify the most characteristic traits of the Neubot design.

3.3 Description of the Design

We started this Chapter by listing, in Section 3.1, the most basic tenets of the Neubot design. Then, in the previous Section, we discussed the more significant challenges entailed by Neubot, thereby adding traits to the Neubot design. Now it is time to organize what we said in the previous Sections, adding to the list presented in Section 3.1. Because such list has eight elements, here we start counting from nine:

9. Neubot shall be flexible (meaning that it shall be simple to deploy new tests and/or to update existing tests) and adaptive (meaning that the host with which to run a test and the test type/methodology depend on the location and on the ISP in which a Neubot instance is);

10. Neubot shall be always-on and shall periodically run network-performance tests;

11. Neubot shall run in the background, may perform long network tests, shall monitor the user activity (for the sole purpose of interrupting tests), and may interrupt a test if it seems to interfere with the user activity;

12. Neubot shall encrypt the communication with its servers to ensure data integrity and authentication of the servers (however, this requirement may be relaxed for the first versions of Neubot);

13. Neubot shall include a privacy policy explaining why it is necessary to collect and publish the user IP address, which are personal data for the European privacy law;

14. Neubot shall be reasonably robust to attacks, not knowing whether a peer-to-peer test is performed with a legitimate Neubot instance or with a malicious one;

15. Neubot instances shall be identified by a random unique identifier, to allow whoever analyzes the data to run time series analysis.

To the above items let us also add the following two requirements:

16. Neubot shall be user friendly and easy to install, moreover it shall allow its users to configure and control it, as well as to navigate its results;

17. Neubot shall be lightweight both in terms of code size and of resources usage.

Having listed the most characteristic traits of the Neubot design, let us go deeper, by describing the components of the Neubot architecture.

3.4 Description of the Architecture

In this Section we describe the architecture of Neubot (see also Figure 3.1). The protagonist is the Neubot instance (often called simply 'the instance'), which has direct or indirect interactions with a number of servers.

3.4.1 Neubot Instance

As discussed above, Neubot shall run in the background. Under Unix systems the Neubot instance is, in fact, a daemon started at boot time. Under Windows the instance may either run as a service (with reduced privileges) or in the context of the user session, depending on the way in which Neubot is installed. In the former case, the instance is started at boot time. In the latter case, instead, the instance is started when the user logs in.

To be more robust to attacks, the Neubot instance should adopt techniques aimed at reducing the privileges as much as possible: at the minimum, the components listening to network requests and performing the transmission tests should not run as root. Also, techniques such as chrooting, sandboxing (of the network experiments), and privilege separation [266] [268] should be considered: i.e., it may be wise (from the security point of view) to run different Neubot components as different processes running on behalf of different users, seeing different portions of the file system, and communicating via pipes.

To communicate with its users, the running-in-background Neubot instance shall listen on the local host for Web API requests. The API shall allow to configure and control the instance (e.g., to run a test immediately) and to request recent results. Optionally, the access to the API could be restricted to applications/users that have authenticated themselves only (e.g., using username and password). As regards the possibility to request tests results, to consume few resources, the instance shall save on the local disk the most recent results only: when results that are not saved on the disk (because they are too old) are requested using the Web API, the instance retrieves them from the Neubot Results Database. The basic way to control the Neubot instance is the Web Interface associated with the Web API. However, different programs (e.g., a system-tray status icon, a command-line application, a graphical user interface) can control the instance via the Web API.

Speaking of resource consumption, to consume few resources the Neubot instance shall typically be idle. By default, it automatically runs tests every 20–30 minutes (the exact time interval between two tests depends on the implementation, however, automatic tests shall never be separated by less than 20 minutes). Periodically, the instance may also monitor the state of the computer and decide whether to run more network experiments, if the computer seems idle and the network seems unused.

Neubot shall be adaptive. In practice, this means that the instance shall consider the test/Test Server suggestions provided by the Neubot servers. In particular: the next test to run is suggested by the Configuration Servers; the Test Server (or Neubot instance[5])

[5]In the following, we may omit for brevity the 'or Neubot instance' part: as we will see, in fact, the

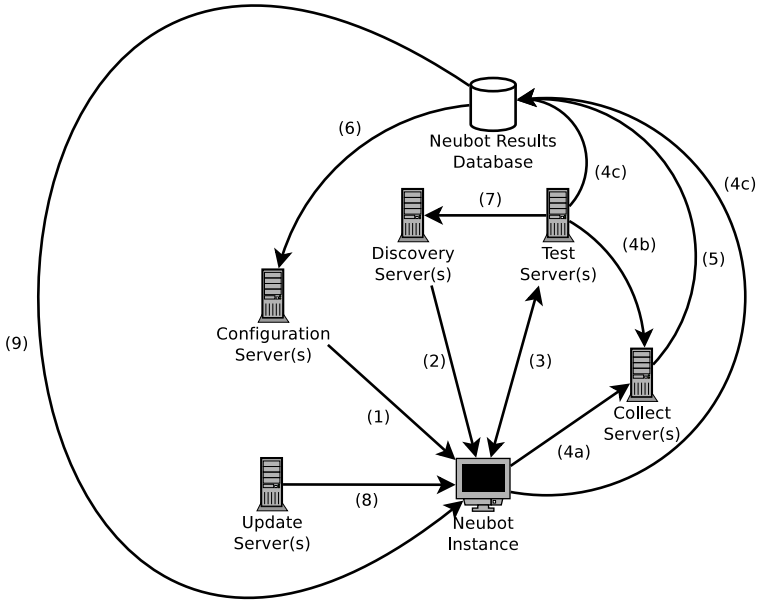


Figure 3.1: The Neubot architecture.

with which run a test is suggested by the Discovery Servers. However, to respect the user preferences, for security and for robustness, the Neubot instance may decide to ignore the suggestions. Moreover: if no Configuration Server is available, the instance shall pick one of the available tests at random; if no Discovery Server is available, the instance shall pick at random one of the well-known Test Servers.

In addition, if the operating system does not allow to automatically upgrade the installed software packages, the instance should periodically check for updates. In turn, this should guarantee flexibility, because new versions of Neubot may add new tests and/or improve existing tests. For increased flexibility, the network tests should be implemented as plugins, packaged separately and updated independently of the Neubot application. Of course, the user shall be able to install, query, disable, enable and uninstall plugins.

Figure 3.1 shows the interactions that an instance has with the Neubot servers. We numbered the interactions and we will refer to the interactions by using their number. The instance: (1) asks the Configuration Server information on which test it should run next; (2) asks the Discovery Server with which Test Server it should run a test; (3) runs a network-performance test with one or more Test Servers; depending on the implementation, may send the test results (4a) to the Collect Servers and/or (4c) directly to the Neubot Results Database; (8) may check for updates by contacting the Update Servers; (9) may download the results of old experiments from the Neubot Results database. In the following Sections we will cover more in detail the role of each server and we will further characterize the

Test Server is a Neubot instance configured in a particular way.

interactions that predates and that follows a network-performance test.

As a final comment, let us add that, since Neubot should work not only in client-server mode but also in peer-to-peer mode, a Neubot instance can also work as a Test Server, if configured to do so. In fact, all the currently-available Test Servers are Neubot instances running on server machines deployed around the world.

3.4.2 Neubot Mobile Instance

The Neubot mobile instance (also called 'the mobile instance') is a Neubot instance that runs on a mobile system. It shall be available for Android [23] and for iOS [67], directly from the platform application store (Google Play Store or App Store). Unlike the Neubot instance (called 'the fixed instance' in this Section), the mobile instance may not run in the background and should only run tests at the user's demand. Moreover, it may not necessarily implement the same tests that the fixed instance implements: some tests may not be implemented and some mobile-specific tests may be implemented. As a final remark, let us add that there shall be a library implementing the functionality commonly used by the mobile and the fixed instances. Such library is described in Section 3.6.

3.4.3 Configuration Servers

The Configuration Servers are servers (one or more) whose task is to provide the Neubot instances with information on the network-performance test (or tests) that the instance should run next (interaction (1) of Figure 3.1).

At the outset, the implementation of the Configuration Servers can be as simple as a routine that picks a test at random from a vector of available tests. As the Neubot architecture evolves, the Configuration Servers should become more complex: in particular, they should periodically fetch Neubot data (interaction (6)) and combine such data with instance-specific information to suggest which test the instance should run next (examples of instance-specific information are: the geographic location of the instance, whether the instance is in a mobile or fixed access network, the access network speed). To tailor the suggestions for the instance, the Configuration Servers need to know the location in which the instance is, therefore they shall implement a mechanism for geolocating it. In addition, if possible, the instances themselves should provide geolocation information.

The result of interaction (1) is that the instance is told which ν tests to run next, with $\nu \geq 1$. A value of $\nu > 1$ may help to reduce the Configuration Servers load and may allow the instance to run independently for quite some time. However, the $\nu = 1$ schema has the significant advantage that the test suggestion returned to the instance may be more up-to-date. The data exchange between the instance and the Configuration Server shall be encrypted. Given the non-persistent nature of the communication, moreover, a request-response protocol such as HTTP is more appropriate to implement the message exchange than a protocol that requires to keep a persistent connection (e.g., XMPP [276] over TCP). As regards interaction (6), it is not specified the frequency with which the Configuration Servers shall fetch up-to-date Neubot data from the Neubot Results Database. As for

interaction (1), also for interaction (6) a request-response protocol is more appropriate than a persistent-connection-based protocol.

Regarding the location of the Configuration Servers, a number of known network locations should be hard-coded in the instance's source code; moreover, there should be the possibility for the instance to update the list of known Configuration Servers using the Update Servers (interaction (8)). In addition, we already said above that - for robustness - if the Configuration Servers are not reachable, the instance should pick a test at random.

3.4.4 Discovery Servers

The Discovery Servers are servers (one or more) whose task is to provide a Neubot instance with the list of IP addresses of Test Servers that are currently available to run network-performance tests. To do that, the Discovery Servers shall also, of course, keep track of the available Test Servers (including the Neubot instances running in peer-to-peer mode that are willing to accept connections from other instances).

A Discovery Server shall allow the connecting instance to choose the criteria used to return the response. In particular, the instance shall be able to request the closest server and a random server. Moreover, it should be possible to request a list of servers that satisfy the instance requirements, rather than a single server. This requirement may be useful not only to reduce the load, but also to implement tests in which multiple Test Servers are involved.

Interaction (2) requests the IP address(es) of one (or more) Test Servers. However, because it may be handy to implement the Configuration Servers and the Discovery Servers together, we do not specify here the exact semantic of this interaction. Because this interaction should only be sporadic, a non-persistent connection is appropriate to implement it. As for interaction (1), encryption is required. As regards interaction (7), it is quite different from interaction (2). Interaction (7), in fact, is used by Neubot instances to register as Test Servers and, while for well-known test servers it may be overkill to keep a persistent connection with a Discovery Server, Neubot instances that occasionally behave as Test Servers may want to use a persistent connection, kept open for the time interval in which they accept incoming connections. Also for interaction (7) encryption is required.

To locate the Discovery Servers, a number of well-known Discovery Servers locations shall be hard-coded in the instance's source code; moreover, there should be the possibility to update the list of Discovery Servers using interaction (8).

3.4.5 Test Servers

The Test Server (in other Chapters sometimes called, for simplicity, 'test server') is - in general - an instance of Neubot, running on a well-known and well-provisioned server configured to run in server mode and to accept connections from other Neubot instances, allowing them to perform network tests. However, also Neubot instances running in peer-to-peer mode may register themselves as Test Servers.

In fact, the Test Server shall be implemented using a specially-configured Neubot instance to pave the way for peer-to-peer tests. However, to implement peer-to-peer tests, the instance shall also implement a mechanism to bypass the restrictions imposed by Network Address Translators (NATs) [285]. To this end, the Neubot instances can also count on the Discovery Servers, which shall also act as STUN (Session Traversal Utilities for NAT) [272] servers.

To register as a Test Server, a Neubot instance shall communicate with one of the Discovery Servers (interaction (7) of Figure 3.1). Because this communication may involve more than one message, it may be appealing to use a persistent, encrypted connection. As regards the communication between an instance and a Test Server (interaction (3) of Figure 3.1), of course, there is no constraint on the type of connection (persistent or not), because it depends on what network protocol the test emulates. As regards the process of saving the network-experiments results, the Test Server may either send the results to the Collect Server (interaction (4b)) or directly to the Neubot Results database (interaction (4c)). In any case, a non-persistent connection is the appropriate choice to move the tests from the machines that run the tests to the ones that collect and store the data.

3.4.6 Collect Servers

The Collect Servers are servers (one or more) whose purpose is to facilitate the collection of the Neubot experiments results. As anticipated in the Test Servers discussion, several strategies to collect the results are possible (including a strategy in which the Collect Servers are not used). When used, the Collect Servers push the results to the Neubot Results Database using interaction (5).

3.4.7 Neubot Results Database

The Neubot Results Database is the place in which Neubot results are saved. It may be implemented by one or more servers, and there may also be multiple Neubot Results Databases. The Neubot Results Database shall implement an API allowing anyone to fetch the experiments results; this possibility, in particular, is exploited by the Configuration Servers and by the Neubot instance (interactions (6) and (9) of Figure 3.1). The Neubot Results Database may also allow any instance to push its experiments results (interaction (4c)). We already discussed the problems entailed by receiving results from arbitrary instances in Section 3.2.1: it is possible to prevent a single Neubot instance from cheating, but much more complex to detect that two instances are consistently reporting fake peer-to-peer tests results.

3.4.8 Auto-Update Servers

The Auto-update Servers are one (or more) servers whose task is to provide Neubot instances with information on available updates, as well as with the binaries that actually implement such updates. These Servers are well-known servers (e.g., releases.neubot.org)

whose location is hard-coded into the Neubot instance sources. They allow to download not only the Neubot releases sources and binary, but also other pieces of information, e.g., the list of Configuration Servers, Discovery Servers, and Collect Servers. Of course, the files that one can download are signed with the Neubot private key, and the Neubot public key is distributed with the Neubot instance, thereby allowing the instance to verify the integrity and the authenticity of the downloaded files.

3.4.9 Master Server

Historically, 'Master Server' (or 'Central Server' or 'Coordinator') was used to indicate the master.neubot.org server, located in Turin and provided by the TOP-IX (TOrino Piemonte Internet eXchange) consortium [141]. Such server, which was the machine on which we used to run the Auto-Update, Configuration and Discovery servers, is still used today to run some services. As such, the Master Server definition is more related to the implementation than it is to the architecture. However, by extension, 'Master Server' is now frequently used with a more-abstract meaning: we use it, in fact, to indicate the group of servers that provide the Auto-Update, the Configuration and the Discovery functionality. We will use this meaning of Master Server, in particular, in Figure 3.2 and in the following Section. To disambiguate, when we want to refer to the specific server machine that we have in Turin, we will use its domain name, i.e., 'master.neubot.org'.

3.5 Description of the Test Sequence

In this Section we describe the test sequence, i.e., the sequence of operations that the Neubot instance performs when it runs a network-performance test. The test sequence is composed of the following operations:

Rendezvous

The Rendezvous is the first phase of the test sequence. The relevant interaction is interaction (a) of Figure 3.2. Historically, the Rendezvous was the name of the API request made by the Neubot instance to master.neubot.org to perform interactions (1), (2) and (8) of Figure 3.1 in one batch. By extension, the name Rendezvous was also used to indicate the test phase in which such API request was made. As the time passed, the architecture and the implementation evolved, but the name of the phase remained. However, now that the Master Server functionality is more distributed, each Rendezvous does not necessarily imply interactions (1), (2) and (8): e.g., the check for updates (interaction (8)) could be performed less frequently than we perform automatic tests (twice per hour); moreover, when the user asks Neubot to run a given test, interaction (1) is not needed.

Negotiate and White-List

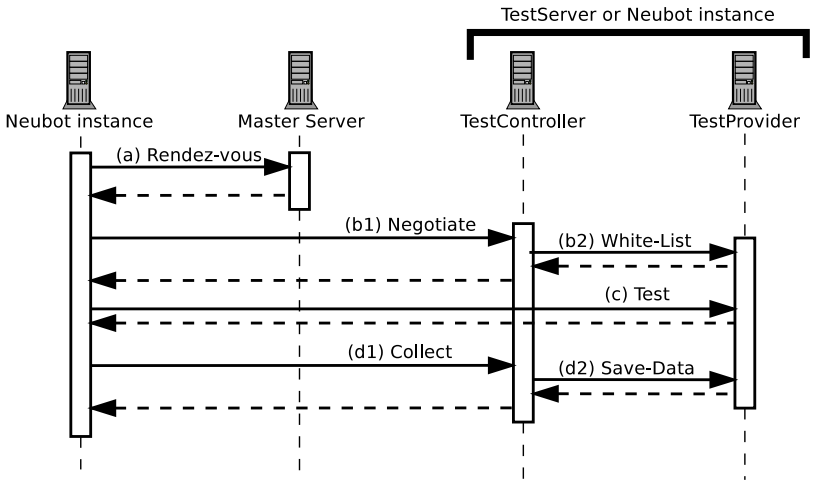


Figure 3.2: Sequence diagram of a generic test.

The Negotiate and the White-List sub-phases make up the second phase of the test sequence. The relevant interactions are interactions (b1) and (b2) of Figure 3.2. In terms of Figure 3.1, the Negotiate and the White-List sub-phases are part of interaction (3), along with all the following phases of the test sequence.

With interaction (b1) the Neubot instance registers to run a test. The TestServer module that takes care of the Negotiate phase is called 'TestController'. Specifically, the Neubot instance sends a TestRequest message containing the test parameters suggested by the instance: e.g., the instance may request a 120-second-long $foobar$ test. The TestController maintains a queue of the Neubot instances that want to run a test: when a pending test completes, the TestController removes the corresponding Neubot instance from the queue; only the first N instances in the queue are allowed to run a test, all the other Neubot instances must wait.

When a pending test completes, the $N + 1$ instance in queue is shifted in position N and is now allowed to run a test. The TestController communicates this fact to the 'TestProvider' (the software module that implements the selected test) with interaction (b2). Specifically, the TestController sends to the TestProvider the White-List message that, among other things, contains the TestRequest message originally sent by the instance. If the TestController and the TestProvider belong to the same process, (b2) is a function call; otherwise, the message may be sent over, e.g., a pipe or an encrypted persistent TCP connection. In Figure 3.2 we show indeed the most general case, i.e., two machines are used.

Upon receiving the White-List message, the TestProvider adds the Neubot instance to its white list (i.e., a list of instances that are allowed to initiate a test). In this phase the test parameters are negotiated: the TestRequest message contains the values proposed by the Neubot instance, and the TestProvider may change them to adapt to its

configuration. For example, if the TestProvider configuration indicates that the maximum test duration is ten seconds and the Neubot instance asks for twenty seconds, the TestProvider will edit the message and reduce the proposed test duration to ten seconds. Then, the TestProvider assembles a TestResponse message that includes: a token that allows the instance to run the test; the (possibly-edited) TestRequest message. The TestResponse message is sent back to the TestController, thereby ending interaction (b2). Successively, the TestController forwards the TestResponse message to the Neubot instance, thereby ending interaction (b1) and the second-phase of the sequence.

Unlike the other interactions seen so far, (b1) and (b2) requires a persistent connection and HTTP may not be the best choice to implement them. As regards the persistent connection, it may be very handy, e.g., to provide intermediate feedback to the Neubot instance. The persistent connection, in fact, could be used to: (i) send the TestRequest; (ii) receive updates such as the Neubot instance position in queue; (iii) receive the TestResponse message. As regards HTTP, if (b1) is implemented as in (i)–(iii), it may not be the most appropriate choice, because there is no clear request-response pattern (more on this topic in Section 3.6.1).

Test

The Test phase is the third phase of the test sequence. The relevant interaction is interaction (c) of Figure 3.2. Typically, the Neubot instance and the Test Server exchange data (usually random data transported with a given protocol) while they measure a number of parameters, e.g., the elapsed time and the download speed.

Collect and Save-Data

The Collect and Save-Data sub-phases conclude the test sequence. The relevant interactions are interactions (d1) and (d2) of Figure 3.2. Each endpoint shares with the other endpoint its measurements, so that both have now perfect information on what happened during the Test phase. Each endpoint concludes this phase of the test by writing on the local storage its own results and the results sent by the other endpoint. In particular, on the server side the Neubot instance message, containing the results of the instance-side measurements, is received by the TestController (via interaction (d1)), which forwards the message to the TestProvider (via interaction (d2)). The TestProvider response, containing the results of the server-side measurements, is then forwarded back by the TestController to the Neubot instance.

To simplify the implementation of the Collect phase, the Test Server may require that the Neubot instance uses a persistent connection to negotiate (as discussed above), so that only Collect attempts sent over such connection by instances running a test are accepted. As an alternative, to implement the Collect using a non-persistent request-response mechanism, the Negotiate phase may return to the instance a ‘single-use cookie’ that the instance can then use to authenticate itself during the Collect.

We conclude the description of the test sequence by describing how a Neubot instance may run in peer-to-peer mode. First, the instance contacts the Discovery Server and communicates its intention to temporarily become a 'Test-Server instance' (interaction (7) of Figure 3.1). The Discovery Server, in turn, will register the instance along with the other available servers, and will suggest the instance to other Neubot instances. Meanwhile, the instance starts to wait for incoming connections (if needed the Discovery Server may help the instance to discovery and traverse a NAT device). When an instance connects to the Test-Server instance, a test takes place, following the procedure described above. At any time, the Test-Server instance may contact the Discovery Server and may communicate its intention to stop being a Test Server (again, interaction (7) of Figure 3.1).

3.6 The Neubot-Library Architecture

In this Section we go more in depth than we did in the previous Sections. We describe, in fact, the software architecture of the Neubot library, i.e., a library that implements the common Neubot functionality, thus facilitating the reuse of such functionality across different platforms (PCs, mobile). We start by discussing a few technical choices that shape the library architecture: which application protocol shall the Neubot instance use to communicate with the Neubot servers; which I/O model shall be used; which programming language shall be used. Next, we describe the main modules of the library.

3.6.1 Choice of the Protocol to Communicate with the Servers

As we have seen in Sections 3.4 and 3.5, most interactions between the Neubot instance and the Neubot servers (described in Figure 3.1 and Figure 3.2) follow the request-response pattern. Moreover, the interaction is often sporadic: there is no need to keep a persistent connection to exchange further messages. Therefore, HTTP (the Hyper-Text Transfer Protocol) and HTTPS (HTTP over SSL) [269] are the best protocols to implement such interactions.

Because HTTP and HTTPS are widely used standards, in fact, their choice brings significant additional benefits. The most obvious (power-)user-side advantage is that, to discover how Neubot works and/or to debug it, one only needs to use a tool such as `curl`. As regards the server-side, instead, the most obvious advantage is that it should be possible to implement the Discovery Servers, and other servers, using a standard LAMP (Linux Apache MySQL PHP) stack [156], as well as using Google App Engine [153], Node.js [118], or simple CGI scripts [148]. This fact opens up the possibility of using several servers, maintained by different parties and using different implementations. For example, should an organization want to run custom Neubot servers, it can implement them using the technological solution that better fits its existing software infrastructure.

As regards the interactions that require persistent connections and full-duplex message exchanges (e.g., interaction (b1) of Figure 3.2), there are two ways to implement them using HTTP. The first possibility is to use the 'Comet' model [149]: the client sends an HTTP

request and the server responds only when the response becomes available. For example, interaction (b1) using Comet could be as follows: the instance sends the TestRequest request and the TestController only responds when the Neubot instance is allowed to run the test. The second possibility is to 'upgrade' the HTTP connection to replace the half-duplex HTTP channel with the bidirectional symmetric channel provided by the widely-used WebSocket protocol [210]. For example, interaction (b1) using WebSocket could be as follows: the Neubot instance sends the TestRequest message, then the TestController sends periodic updates concerning the Neubot instance position in queue, finally the TestController forwards the TestResponse message to the Neubot instance.

3.6.2 Choice of the I/O Model

Because Neubot shall consume as few resources as possible and because Neubot may run network performance tests with multiple Test Servers (or Neubot instances), the Neubot library shall use a scalable I/O model. Clearly, this requirement rules out the traditional 'blocking' I/O model by which two threads are created per TCP connection, one for receiving and the other for sending data. For an application that aims to consume few resources, in fact, it is not acceptable to create two threads per TCP connection.

The traditional model is called 'blocking' because a thread is not runnable when a slow network operation is pending: e.g., if there is no available data, the 'recv' system call waits until new data is received. More scalability can be achieved by making recv (and other slow system calls) 'nonblocking' (i.e., recv will return an error if there is no data in the socket buffer) and by using a mechanism to notify the application when the socket can be read and/or written. This model is more scalable than the blocking model because a single thread may serve a number of concurrent connections. The price to pay for scalability is that the application shall be structured to react to events (e.g., 'it is now possible to read from the socket'): in its simplest form, the application is structured around a 'reactor' loop [164] [198] that monitors the state of many sockets and, whenever the state of a socket changes, invokes the proper registered-in-advance handler function.

There are many mechanisms to inform the application that the socket state has changed, as explained by the well-known blog post by Dan Kegel "The C10K problem" (where C10K means '10,000 concurrent connections') [233]; in the following we describe the one that, to the best of our knowledge, is the most commonly used: the 'nonblocking' I/O model with 'level-triggered readiness notification'. In its simplest form, such model works as follows: the reactor loop continuously invokes the 'select' system call passing it a list of sockets to monitor. In turn, the select does not return until the state of one-or-more sockets change. At that time, select returns a list of readable and/or writable sockets (meaning respectively, that there is data in the receive buffer and that there is empty space in the send buffer). Such list is then used by the reactor as follows: it invokes the read-handler function of each readable socket and the write-handler function of each writable socket.

This model (nonblocking I/O with level-triggered readiness notification) is implemented by a number of libraries and frameworks, including, e.g., Libevent [75] and Twisted [13]. It is a good candidate I/O model for Neubot, also because it has been implemented in a

number of different languages. However, it is not the only possible I/O model; there is, in fact, another I/O model that may be suitable: the lightweight-threads-based I/O model.

The lightweight-threads-based I/O model is implemented by a few high-level programming languages, such as Erlang [46] and Go [52]. The general idea is that the language implements lightweight application-level threads (LWTs) mapping a number of LWTs onto a single operating system thread. In its backend the language may use a nonblocking I/O model to efficiently handle multiple sockets, but such complexity is hidden to the programmer. In fact, following the 'actor model' [147], the programmer only needs to define small well-defined 'actors' that receive messages and act accordingly (e.g., by sending replies or by creating more actors). This second I/O model is very promising: Erlang, e.g., allows one to create very-high-concurrency HTTP servers [25]. However the main drawback is that such model is tightly coupled with the specific high-level programming language.

Having evaluated both I/O models, we decide to use the nonblocking I/O model because, being Neubot a network-performance measurement tool, we need control over the handling of the sockets. Arguably, in fact, we have more control if we use a library that simplifies nonblocking I/O (e.g., Libevent) than if we use a language hiding the whole I/O management strategy behind the LWTs curtains (e.g., Erlang).

3.6.3 Choice of the Programming Language

Because Neubot shall run on both PC and mobile platforms, its library shall ideally allow the different implementations to share bits of code. However, different platforms use different reference languages (e.g., iOS [67] applications are written in C/ObjC, Android [23] applications are written in Java [69], PC applications are written using a vast array of languages, including C/C++ [29] and Python [130]); therefore, Neubot for a platform will always contain platform-specific code written using the reference language of that platform. Still, the majority of the code is unlikely to be platform specific; therefore, one may be tempted to write the library using a high-level programming language (e.g., Python, Lua [80]) that is available on the target platforms. For example, if the Python language is chosen: one can compile CPython (i.e., the default Python implementation, written in C) [130] and bundle it with the Neubot iOS package; Jython (an alternative Python implementation written in Java) [71], instead, can be used on Android; CPython can be used on PC platforms.

The problem with using a high-level programming language, however, is that a mobile platform may not have enough resources to efficiently run network-performance tests written using such language. Not knowing in advance on which kinds of devices is Neubot going to run, therefore, we prefer a mixed approach in which performance-critical routines are written in C/C++. The C/C++ family, in fact, is the most-efficient family of languages available on all platforms (including Android, in which there is the native development kit, NDK [24], that allows to use libraries written in C/C++).

The mixed approach guarantees efficiency but raises the problem of how to glue the high-level code and the C/C++ code. However, such problem is already solved by tools like SWIG (the Simplified Wrapper and Interface Generator) [182] [137] and Libffi (the

Foreign Function Interface library) [76]. SWIG, in fact, allows one to generate C/C++ code allowing high-level code to call C/C++ code, and Libffi allows one to call code written in one language (e.g., C/C++) from another language (e.g., Python).

Having solved the problem of gluing languages together, we now face another design problem: which functionality shall be implemented in C/C++ and which shall be implemented using the chosen high-level programming language? The obvious response is that the code dealing with the network and with processing network messages shall be implemented in C/C++, because in C/C++ it is possible to write efficient event-processing loops (consider, e.g., Libevent [75] and Libuv [78]) and zero-copy network-protocol parsers (consider, e.g., the 'HTTP parser' library [65] used by Node.js [118]).

As far as the processing of network messages is concerned, we can, therefore, refine our question as follows: which is the most appropriate boundary between the C/C++ code and the high-level code? To shape our answer, let us remind that very often the data that traverses the high-level-language–C/C++ boundary is copied. In turn, the copy may reduce the performance, if the copied data size is large. For example, if we ask SWIG to generate Java native interface (JNI) [155] wrappers for the following C code:

```
void write_string(char *buf);
```

we obtain the following (slightly edited) wrapper code:

```
SWIGEXPORT void JNICALL Java_ExampleJNI_write_1string(JNIEnv *jenv,
    jclass jcls, jstring jarg1) {
  char *arg1 = (char *) 0;
  if (jarg1) {
    arg1 = (char *)(*jenv)->GetStringUTFChars(jenv, jarg1, 0);
    if (!arg1) return;
  }
  write_string(arg1);
  if (arg1) (*jenv)->ReleaseStringUTFChars(jenv, jarg1,
                       (const char *)arg1);
}
```

in which the 'GetStringUTFChars' function is invoked. Now, if we check how such function is implemented[6] by Dalvik [41], the Java virtual machine (JVM) used by Android, we see that the returned string is a copy of the string provided as argument (as hinted by the value that is assigned to the 'isCopy' flag):

```
static const char* GetStringUTFChars(JNIEnv* env, jstring jstr,
  jboolean* isCopy) {
    ScopedJniThreadState ts(env);
```

[6]The code, copyrighted by the Android project and released under the Apache 2.0 license, was taken from the Dalvik Git repository [48] on January 19, 2013 and was edited to increase readability.

```
    ...
    if (isCopy != NULL) {
        *isCopy = JNI_TRUE;
    }
    StringObject* strObj = (StringObject*) dvmDecodeIndirectRef(
                           ts.self(), jstr);
    char* newStr = dvmCreateCstrFromString(strObj);
    ...
    return newStr;
}
```

Therefore, the string that 'write_string' receives is always a copy of the string that the high-level language (in this case Java) passed to the 'write_string' (JNI) wrapper. Abstracting what we learned from this example, we can now reply to the above question: *the boundary between C/C++ and the high-level language shall be such that we avoid unnecessary data copies*. For example, to measure the HTTP download speed, the high-level code may ask the lower level code to retrieve a given URL, to return the HTTP response headers, to measure and return the download speed, but to discard the HTTP response body; thereby avoiding the expensive unneeded-at-higher-level copy of the response body. Another possible solution is the following: allocate objects (e.g., buffers) in the high level language and make sure that such objects are not automatically garbage collected by the high level language when they are being used by the low level code. The disadvantage of the latter solution is that such code needs to be partially written at hand (one actually needs to configure SWIG to automatically generate the desired code).

To wrap up: Neubot shall be available both on PC and mobile platforms and the Neubot library shall allow to share code between different platforms. To this end, the library may be written using a to-be-defined high-level programming language (e.g., Python, Lua) that is available on the target platforms. However, for performance reasons, the networking code of the library shall be implemented in C/C++. The boundary between the low-level code and the rest of the library shall be such that unnecessary copies of data are avoided.

3.6.4 Library Modules

In this Section we describe the modules of the Neubot library. The diagram in Figure 3.3 shows the modules architecture. The internal yellow wheel represents the basic modules. The external red wheel represents helper modules that should be implemented on top of yellow-wheel modules. We describe now the modules that make up the Neubot library, following the order in which they appear in Figure 3.3:

Host load

This module is responsible of measuring the host load. The load is measured in terms of: memory usage, number or running processes, load average, network usage.

Optionally, the module shall also check whether well-known applications (e.g., BitTorrent [27]) are running. The module allows one to take a snapshot of the current host load, as well as to determine whether the host load significantly increased since the previous snapshot. Measuring the host load is a precondition to run long network measurements. If the host load is now known, in fact, Neubot can only run short tests, to avoid depleting the user's foreground flows. Knowing the host load, conversely, Neubot can run arbitrarily-long tests: e.g., a long test can be started when the host is idle and may be terminated if the user starts new possibly-bandwidth-hungry network flows. Moreover, the host load also allows one to assess the impact of such load onto the measured network performance.

Plugins

The Neubot library shall also allow Neubot to manage its network tests as independent plugins. Managing its tests as plugins, Neubot can adapt itself to the context: when the test methodology of a given test needs to be changed, this can be done by releasing a new version of the corresponding plugin. Moreover, the a-test-is-a-plugin design allows us to extend Neubot: when a new test is added, we update the available-plugins list on the Update Servers, and we wait for the Neubot instances to install the new plugin.

Let us now frame the discussion of the plugins functionality in terms of the programming language choice. When we discussed such choice in Section 3.6.3, we said that part of the library was to be implemented using a high-level programming language, π, and part was to be implemented in C/C++ for performance reasons. The question, therefore, is whether the plugins shall be implemented only in π, only in C/C++, or both. For ease of implementation one may be tempted to vote 'π only'. That solution works, however, as long as the low-level functionality required by a plugin is provided by the C/C++ part of the library. It may happen, however, that a new plugin requires a functionality that is not provided by the C/C++ part of the library. Therefore, it is more robust to require that the library can deal with both plugins written in π and plugins written in C/C++.

HTTPRouter

The HTTPRouter module allows to register a handler function that serves HTTP requests. The proper handler is selected by checking the HTTP host header and the URI path of an incoming HTTP request. The HTTPRouter shall be implemented in C/C++: parsing network messages may be, in fact, performance critical (as discussed in Section 3.6.3). The HTTPRouter implementation shall allow the programmer to choose the operating mode, as follows. In message-oriented mode, the handler is called when the whole message is received. In streaming mode, the handler is called: when the headers are received, when each part of the body is received, and at the end of the body. In discard mode, the handler is called when the headers are received and at the end of the body. The message-oriented mode is convenient for reading small

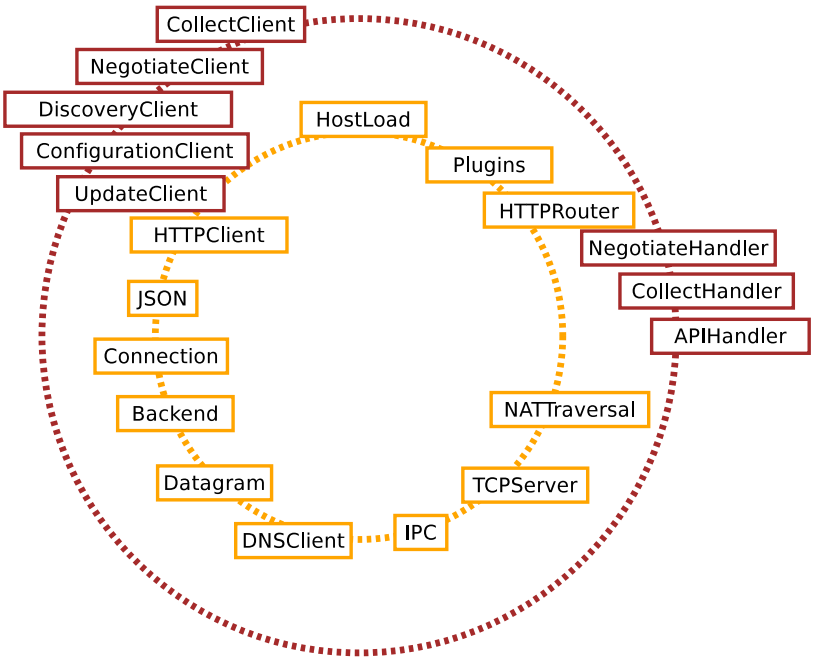


Figure 3.3: Neubot library.

messages (such as the ones exchanged before and after a test) and shall also allow the programmer to set the maximum-accepted message size. The streaming mode allows one to read arbitrarily large bodies without consuming too much memory. The discard mode allows a handler written in a high-level programming language to avoid unnecessary copies (as discussed in Section 3.6.3).

The HTTPRouter module shall be used to implement the Test Server functionality (see Figure 3.1). Because this functionality is part of the Neubot library, it allows a Neubot instance to become a Test Server. In particular, the HTTPRouter shall be used to implement the Negotiate Handler, the Collect Handler, and the Data Handler, which are discussed below.

NegotiateHandler

The Negotiate Handler module is used by the Neubot instances running as Test Servers. It implements part of the TestController functionality described in Figure 3.2. The Negotiate Handler runs on top of the HTTPRouter; it may be implemented using a high-level language; it processes negotiate requests from Neubot instances. The relevant interactions, in particular, are interaction (b1) and (b2) of Figure 3.2.

Collect Handler

The Collect Handler module is used by the Neubot instances running as Test Servers. It implements part of the TestController functionality described in Figure 3.2. The Collect Handler runs on top of the HTTPRouter; it may be implemented using a

high-level language; it receives collect requests from Neubot instances. The relevant interactions, in particular, are interaction (d1) and (d2) of Figure 3.2.

APIHandler

The APIHandler module serves Web API requests. The APIHandler is implemented on top of the HTTPRouter, and may be implemented using a high-level language. As explained in Section 3.4.1, the API allows the user of the Neubot instance to configure the instance; to ask the instance to run an on-demand test; to navigate the test results. As regards the test results, in particular, when the results of old tests that are no longer saved locally are requested, the APIHandler will retrieve such results from the Neubot Database Servers (interaction (9) of Figure 3.1).

NATTraversal

The NATTraversal module is a fundamental pre-requisite to implement peer-to-peer measurements, because it allows a Neubot instance behind a firewall to become a Test Server. The NATTraversal module shall employ techniques such as STUN (Session Traversal Utilities for NAT) [272] to determine the type of NAT [285] device (if any). To bypass the NAT, the NATTraversal module shall try to cooperate with the home router (if possible), for example using the Internet Gateway Device Protocol (IGD) [187] to request the home router to 'open' one or more network ports for Neubot. If IGD is not available, the NATTraversal module may try to use other NAT traversal techniques, e.g., hole punching [215].

TCPServer

The TCPServer is a C/C++ module that allows one to create a TCP server that listens on one-or-more address-port pairs. The TCPServer module shall also allow one to register a handler function ϕ. The handler function ϕ is invoked when a new connection γ is accepted. It is ϕ's responsibility to setup the context of connection γ and dispatch the newly-created connection to the proper protocol-specific software module (e.g., if a new connection is created for the BitTorrent protocol, ϕ passes such new connection to the BitTorrent-protocol module).

IPC

The Neubot library shall include a module to facilitate IPC (Inter Process Communication). The IPC functionality may be useful if we decide to implement the TestController and the TestProvider functionality of Figure 3.2 using two separate processes on the same machine: in such case, interactions (b2) and (d2), rather than being simple function calls, will be performed using an IPC mechanism. Another case in which IPC is needed is the one in which, for additional robustness and security, the Neubot instance is implemented using multiple communicating processes with different privilege levels, as discussed in Section 3.4.1.

On UNIX and Windows, despite the many differences, the IPC functionality may be implemented using pipes. However, the IPC functionality may also be more easily

implemented using BSD sockets, which have (approximately) the same behavior on both platforms. Moreover, BSD sockets may also allow, with a little more work, to have processes that communicate across machines. Such feature is interesting because it paves the way for a more scalable architecture: e.g., a single TestController may control a number of TestProviders deployed on different machines.

DNSClient

The DNSClient is a module that allows to make Domain Name System (DNS) [152] requests and to process the corresponding DNS responses. The DNSClient shall allow one to use both the default DNS server and a specific DNS server (e.g., the public Google DNS server [53]). The DNSClient is part of the Neubot library architecture because, as discussed in Chapter 1, DNS filtering and DNS tampering are two key techniques to prevent access to resources. Therefore, the presence of the DNSClient paves the way for network experiments concerned not only with network performance, but also with freedom of expression (see, e.g., the MorFEO project [91]).

Datagram

The Datagram module is a C/C++ module that provides the abstractions needed to deal with UDP sockets: it allows to create, bind and connect datagram sockets, as well as to send and receive messages. This module is the basic building block that one can use to construct network-performance tests that use UDP.

Backend

The Backend is the Neubot-library module responsible of storing data on the local-machine filesystem. It allows the programmer, in particular, to load and store recent test results and the Neubot settings. As regards the results, the Backend shall implement a circular queue, to guarantee that only the most recent results are stored on the disk. In turn, this allows Neubot to keep under strict control the disk space used to store results.

Connection

The connection is a C/C++ module that provides an abstraction, Σ, representing a stream-like network connection. As such, it is suitable to represent, e.g., a connected TCP socket (but possibly also a UNIX-domain socket). Of course, the Neubot library shall implement and export the Σ abstraction because it is the basic building block to construct TCP-based network-performance tests.

The features that Σ shall implement are the following. One shall be able to establish a TCP connection by providing Σ with the protocol family, the address and the port number. Moreover, the programmer shall be able to initialize Σ with a file descriptor. Additionally, once Σ is attached to a file description, it shall be possible to establish a SSL connection. In addition to the common methods that allow to receive and send data, Σ shall also export methods that allow to read a line, to read an exact amount of bytes and to discard an exact amount of bytes.

All the features described above are useful to efficiently implement network-performance tests. In particular, as discussed in Section 3.6.3, the functionality that allows to discard an exact number of bytes is useful to reduce the impact on performance of data copying when a test is implemented using a high-level language.

JSON

The Neubot library shall also allow one to serialize and deserialize objects using the JavaScript Object Notation (JSON) [70] representation. This functionality is needed to process the results of many Neubot messages, which are encoded using this format. We chose this format over the Extensible Markup Language (XML) [167], because JSON allows one to encode easily the most commonly used data types: numbers, strings, vectors and dictionaries. Moreover, being JSON textual and human-readable, it is simple to understand thus simplifying debugging. However, because JSON cannot easily transport binary data, there may be cases in which other data representations are used.

HTTPClient

The HTTPClient module, written in C/C++, is the basic building block to construct network-performance tests based on HTTP, as well as to construct helper modules that simplify the interactions between a Neubot instance and the Neubot Servers (see Figure 3.1 and Section 3.4). As discussed in Section 3.6.1, in fact, HTTP is the default protocol used by Neubot for communicating with the Neubot servers.

The HTTPClient module shall allow to connect to a given HTTP server and issue multiple (possibly pipelined) requests using the same, persistent TCP connection. The HTTPClient module shall also allow one to specify an handler function that processes the response(s). Moreover, the HTTPClient shall allow the programmer to choose the operating mode, as follows. In message-oriented mode, the handler is called when the whole message is received. In streaming mode, the handler is called when any of these events occur: the headers are received, when each part of the body is received, and at the end of the body. In discard mode, the handler is called when the headers are received and at the end of the body. The message-oriented mode is convenient for reading small messages (such as the ones exchanged before and after a test) and shall also allow the programmer to set the maximum-accepted message size. The streaming mode allows one to read arbitrarily large bodies without consuming too much memory. The discard mode allows a handler written in a high-level programming language to avoid unnecessary copies (as discussed in Section 3.6.3).

UpdateClient

Based on the HTTPClient and possibly written using a high-level programming language, the UpdateClient is responsible of checking for (and for downloading) updates from the Update Servers (interaction (8) of Figure 3.1). Once an update is downloaded, the UpdateClient verifies its authenticity and integrity, and copies it on the

filesystem. Afterwards, the UpdateClient takes the proper steps to ensure that the downloaded update is used. For example, if a new plugin was downloaded, the UpdateClient will inform the Plugins module that such plugin was added or upgraded.

ConfigurationClient

Based on the HTTPClient module and possibly written using a high-level programming language, the ConfigurationClient allows one to query the ConfigurationServers (interaction (1) of Figure 3.1) to retrieve information on the next tests to run. This functionality is implemented into (and exported by) the Neubot library to simplify the task of writing plugins.

DiscoveryClient

Based on the HTTPClient module and possibly written using a high-level programming language, the DiscoveryClient allows one to query the DiscoveryServers (interaction (2) of Figure 3.1) to retrieve the address (or the addresses) of the Test Server(s) with which a test shall be run next. This functionality is implemented into (and exported by) the Neubot library to simplify the task of writing plugins.

NegotiateClient

Possibly written using a high-level language, this module hides the complexity of the Negotiate (interaction (b1) of Figure 3.2). This functionality is implemented into (and exported by) the Neubot library to simplify the task of writing plugins.

CollectClient

Possibly written using a high-level language, this module hides the complexity of the Collect (interaction (d1) of Figure 3.2). This functionality is implemented into (and exported by) the Neubot library to simplify the task of writing plugins.

With this description of the modules of the Neubot library we conclude this Chapter about the Neubot architecture. But, before we move to the next Chapter (in which we describe the current Neubot implementation), let us draw the conclusions.

3.7 Concluding Remarks

In this Chapter we described the Neubot architecture. The cornerstone of the architecture is the open-source Neubot software, that shall run on PCs and mobile platforms. Once installed, Neubot shall run multi-protocol network-performance tests with test servers or, in peer-to-peer mode, with other Neubot instances. The results will be released as open data and will be useful to study broadband performance and, more specifically, network neutrality.

Neubot shall not run passive measurements, because they are not compatible with the Neubot open-data policy. Rather, Neubot should only run active network performance

tests, in which existing protocols are emulated and random data is transferred. Speaking of privacy, Neubot shall also include a privacy policy explaining what personal data needs to be processed and why. In particular, the privacy policy shall asks the permission to collect and publish the user's IP address, a personal data piece of data according to European privacy laws, which is crucial to study the collected data. The IP address, in fact, allows a researcher to map the experiments results to an ISP and to an approximate geographic location.

Neubot shall be flexible and adaptive. Flexible because its tests shall be packaged as plugins that can be installed and updated independently of the Neubot application itself. Adaptive because the information on (a) which test to run and (b) with which server (or Neubot instance running in peer-to-peer mode) will be suggested to Neubot by the Neubot servers. In particular, the Neubot servers shall tailor the returned suggestions taking in consideration information such as the location and the ISP of a Neubot instance.

Neubot shall run in the background. This fact, coupled with a Neubot module that shall allow to measure the load (CPU, network, memory) of the hosting computer, should allow Neubot to run arbitrarily-long tests that do not clash with the user activities. Moreover, because peer-to-peer test are run with unknown possibly-malicious other Neubot instances, the Neubot implementation shall be reasonably robust to attacks. To this end, it shall drop the privileges and it may confine the network-tests code into sandboxed processes.

In addition to the Neubot instance, the Neubot architecture is composed of the following servers: Configuration Servers, Discovery Servers, Test Servers, Collect Servers, Neubot Database Servers, Auto-Update Servers. The Configuration Servers and the Discovery Servers allow a Neubot instance to discover, respectively, which test to run next and with which Test Server (or Neubot instance). The Test Servers are used to run the tests. Note that, because Neubot shall run peer-to-peer tests, any Neubot instance may become a Test Server. Collect Servers facilitates the process of collecting the Neubot experiments results. Neubot Database Servers store the tests results and exports them using a Web API. Auto-Update Servers allow Neubot instances to stay up to date.

The automatic test sequence, run by each Neubot instance every 20-30 minutes, is composed of the following phases: at first, the Configuration Servers, the Discovery Servers and, possibly, the Update Servers shall be contacted; at the end of this phase, the Neubot instance should know the test to run and the Test Server (or Neubot instance) with which the test shall be run; next, the Neubot instance shall negotiate the test parameters with the Test Server (possibly waiting for its turn to run the test); afterwards there is the actual test; finally, the Test Server and the Neubot instance shall exchange the measurements taken during the test and save them on their local disk.

The Neubot library, used to share code between the PC and the mobile Neubot implementations, shall be partially written in C/C++ (for performance) and partially in a to-be-specified high-level language (possibly Python or Lua). Among the many modules of such library, there shall be modules that facilitate the usage of HTTP, which is the protocol that Neubot shall use by default to communicate with its servers. The Neubot library shall be based on the 'nonblocking I/O with select-based readiness notification' model, because it provides a good compromise between scalability (and resources usage), on the one hand,

and control over the way in which sockets are handled, on the other hand.

Chapter 4

Description of the Implementation

In this Chapter we describe the current implementation of Neubot, using as reference the Neubot architecture introduced in the previous Chapter. This description gives us the opportunity to strike a balance of the Neubot implementation status, given that – as mentioned in the previous Chapter – not all the components and the modules of the Neubot architecture are fully implemented yet (in particular, peer-to-peer tests and arbitrarily-long tests are not yet implemented). As we will see, however, despite a number of features were not ready, the current Neubot implementation has been already flexible enough to allow us to run the advanced experiments described in the following Chapters. We will conclude this Chapter with a description of the features on which we are working, to give the reader a sketch of the future trajectory of Neubot as a tool and as a project.

The remainder of this Chapter is organized as follows. We start with Section 4.1, in which we describe the current implementation status, using the previous Chapter as reference. Afterwards, in Sections 4.2 and 4.3, we provide more details on the currently-implemented tests and the currently available servers. Finally, in Section 4.4 we draw the conclusions and describe future work.

4.1 Implementation Status

In this Section we provide an overview of the current Neubot implementation (Neubot 0.4.16.9 released on October, 2013). We will frame the discussion in the context of the description of the Neubot architecture presented in Chapter 3. In other words, we will report on the implementation status of the components and modules described in Chapter 3.

Available on PC platforms only

The current version of Neubot is written in Python 2.x and is available for PC platforms only (i.e., Intel- and AMD-based desktop, laptops and netbooks). Non-blocking I/O is implemented using the Python select [127] and socket [128] modules. Python was chosen over other high-level languages because of its clear syntax and because of its feature-rich standard library. Neubot is packaged for: Windows

XP+, Debian GNU/Linux and other Debian-based distributions; MacOSX 10.6+; FreeBSD -current. Auto-updates are implemented for MacOSX and Windows only: the other systems already implement the automatic update of installed packages, therefore, there is no need to duplicate such functionality. On Windows, we distribute a py2exe [12] frozen executable[1]. On other platforms, we distribute the Neubot Python modules and we declare Python 2.x as a runtime dependence. We are currently working to port Neubot to the Android platform [23]. The Neubot Android application will be based on a Neubot library partially-written in C/C++ (as explained in Section 3.6) and partially written in a yet to-be-defined high-level programming language. As regards the latter, we have two candidates: Lua [80] and Python [130]. The advantage of using Lua would be that Lua is efficient and easy to embed. Moreover, it seems easy to use Lua to create a sandbox that prevents the high-level code to perform certain operations (e.g., open filesystem files). As regards Python, the main advantage is, of course, that Neubot is already written in Python.

Client-server tests only

Neubot does not implement peer-to-peer tests yet. The code can, in theory, run in peer-to-peer mode (in fact, the same code tree [49] implements both the Neubot instance and the Test Server), but minor tweaks are needed to run a Test Sever on Windows, because the Test-Server-specific code is written with Unix in mind. More critically, to properly implement the peer-to-peer tests functionality, two features are missing: the infrastructure support to register a Neubot instance as a Test Server is not yet implemented; the NATTraversal module, needed allow a Neubot instance to bypass a Network-Address-Translator (NAT) [285] device, is not yet implemented.

Integrated with M-Lab

Since version Neubot 0.4.6 (released on January 24, 2012), Neubot is hosted by M-Lab [206], a consortium – described in Chapter 2 – that hosts the test servers of open network-performance measurement tools such as Neubot. In Section 4.3, in which we describe the implementation of the Neubot Servers, we will further characterize the integration of Neubot and M-Lab.

Open source and open data

Neubot is open-source and Neubot data is published as open data. The source code is released under the terms and conditions of the GNU General Public License v3 [55]. The Neubot repository is available at GitHub [49]. Neubot data is available under a

[1]Py2exe is a Python extension that allows to transform a set of Python modules into a Windows executable: the executable is linked with the Python DLL (as well as with other helper DLLs) and runs the entry-point Python module. The other modules (as well as all the standard library modules that are needed) are distributed along with the executable as bytecode-compiled Python files. The result of this process is called 'frozen executable' because the set of modules and all their dependencies are gathered and bundled together into a single package, thereby 'freezing' the interface between the modules.

Creative Commons Zero 1.0 waiver [30]. Mirrors of the Neubot data are available at the Google Cloud Storage [36] and at the data.neubot.org [103] web sites.

Four active network-performance tests are implemented

Neubot 0.4.16.9 implements four active network-performance tests (described in Section 4.2): 'HTTP Speedtest', 'BitTorrent', 'raw' (or 'raw test') and 'Dashtest'.

Pure-Python plugins are supported

The Dashtest test is the first Neubot test written as a pure-Python plugin [89]; the other plugins, in fact, are part of the Neubot core. The Dashtest plugin, in particular, only depends on a few Neubot-core modules that roughly correspond to the Neubot-library modules described in Section 3.6. However, the work to package and deploy plugins independently of the Neubot core has not started yet. Similarly, it has not started yet the work to sandbox network experiments to increase robustness.

Automatic updates are implemented

Neubot implements automatic updates on MacOSX and Windows, the two supported systems for which they were needed. The updates are downloaded using HTTP, but the update tarball (containing a new version of Neubot) is signed using the Neubot RSA private key. Before automatically installing the updates, for increased security, a Neubot instance verifies the signature of the update tarball using the companion RSA public key that is part of the Neubot MacOS and the Neubot Windows distributions.

Neubot uses a static test-selection policy

The current version of Neubot does not tailor the test policy depending on the Neubot-instance geographic location and/or ISP yet. In fact, the Configuration Server is currently not implemented. To select the next test, instead, a Neubot instance picks one of the available tests at random. Not all tests are selected with the same likelihood, however. In fact, we recently tweaked the probability of selecting a test to favor the Dashtest, because we needed to collect more Dashtest points for the MMSys 2014 paper [179]. Currently, the 'raw' test is selected with 9% probability, the BitTorrent and the HTTP Speedtest tests are selected with 27% probability each, the Dashtest is selected with 36% probability. As regards the proper implementation of a Configuration Server, we are working on a side project, called 'Neubot visualizer' (Neuviz, see Chapter 10), which aims at performing on-the-fly analysis of the Neubot results, to provide information useful to tailor the test policy.

Neubot runs in the background

Neubot runs as a system daemon on Unix; in the context of the user session on Windows. The Unix daemon runs as the unprivileged '_neubot' user. The code that drops privileges was double checked with the code of MacOSX launchd [72] as well as with the OpenBSD 'daemon' system call [119]. Unlike other Unixes, Neubot for MacOSX uses two daemons: in addition to the unprivileged daemon, there is a also a

privileged daemon that implements the auto-update functionality [105]. Such daemon does not directly access the network, rather, it spawns an unprivileged child, running as '_neubot_update'. The parent and the child communicate via a pipe: the child downloads the file indicated by the parent and returns the downloaded file over the pipe. The parent does not trust the child: the status messages returned over the pipe are always checked with regular expressions; moreover, as mentioned above, the downloaded files are also checked using the public RSA key distributed along with Neubot. As regards Windows, given the complexity of dropping the privileges on Windows (which probably stems from our ignorance of the platform), we decided for now to run Neubot in the context of the user session. Therefore, Neubot is installed in the user home directory, it is started when the user logs in, and it is killed when the user logs out.

Neubot runs short tests only

Neubot does not implement a mechanism to measure the user activity (called HostLoad in Section 3.6) yet: we only have a prototype HostLoad for Linux, which currently is not used. Therefore, to avoid clashing with the user's foreground flows, all Neubot tests have limited duration. For this reason, when in the following Chapters we will provide a thorough description of the Neubot tests implementation, we will always underline which strategy is used to avoid clashing with the user's flows.

Neubot uses HTTP by default

Neubot uses HTTP for all the message exchanges, except – of course – the network tests. When there is no clear request-response semantic and when the connection shall be persistent (e.g., during the Negotiate phase of the test), Neubot uses Comet [149] to emulate a full-duplex persistent channel using HTTP.

No encryption is used

We do not encrypt the message exchanges between a Neubot instance and the Neubot servers yet. However, as we pointed out above, we use RSA keys to check the authenticity and integrity of the downloaded automatic updates.

A privacy policy is in place

Neubot has a privacy policy [101]. Consistently with the European and the Italian privacy laws [1], and because – as explained in Section 3.2.1 – Neubot needs to collect the IP address of users (which in the EU is considered personal data [5]), the privacy policy (i) explains why Neubot needs to collect and publish the user's IP address and (ii) asks the Neubot user for the permission to collect and publish his/her IP address. Neubot does not perform any test until the user (a) confirms that he/she has read the policy and (b) provides the permission to collect and publish his/her IP address.

A unique random identifier is used

Each Neubot instance is identified by a random unique identifier, suitable to perform time series analysis (as discussed in Section 3.2.2). The random unique identifier does not represent personal data, therefore, there is no need to mention it in the privacy policy. However, the online FAQ [100] mentions the existence of such unique identifier and explains why it is needed. For the paranoid users, there is also a command line command that allows one to generate a new random unique identifier.

Neubot has a local Web user interface

Neubot implements an AJAX-based local Web user interface [113], shown in Figure 4.1, that uses the localhost-only Neubot-instance API [104] to interact with the local Neubot instance. The Web user interface allows the user: to check the status of the Neubot instance, to run on-demand tests, to change the settings, to read and accept the privacy policy, to see recent results, to read the logs. To simplify the task of presenting the results of network tests, we designed a mini-language [107] [106] inspired to Lisp [157] that encodes the sequence of transformations needed to generate plots and tables from the results into a JSON [70] object. For example, the following piece of JSON

```
["to-millisecond",
  ["reduce-avg",
    ["select", "alrtt_list",
      ["select", "client",
        ["parse-json",
          ["select", "json_data", "result"]]]]]]
```

means 'select the json_data field of the each test result, convert it to JSON, take the client field, take the alrtt_list field and compute the average of the list, convert the result to millisecond' and is used to compute the average application-level RTT from the vector of ten measurements included in the 'raw' test results.

After this overview of the Neubot implementation, let us discuss some aspects more in depth, starting with the implemented tests.

4.2 Implemented Network-Performance Tests

In this Section we provide a brief description of the four active network-performance tests implemented by the current version of Neubot (0.4.16.9).

Before we describe the tests, however, let us discuss how the interval between two automatic tests, δ, is computed. At startup, Neubot selects δ at random within the 23–27 minutes interval, to avoid undesirable Neubot-instances synchronization. To understand why synchronization is undesirable, let us assume that δ is fixed and equal to 25 minutes. Let us also assume that there are 100 Neubot instances waiting to run a test with a given

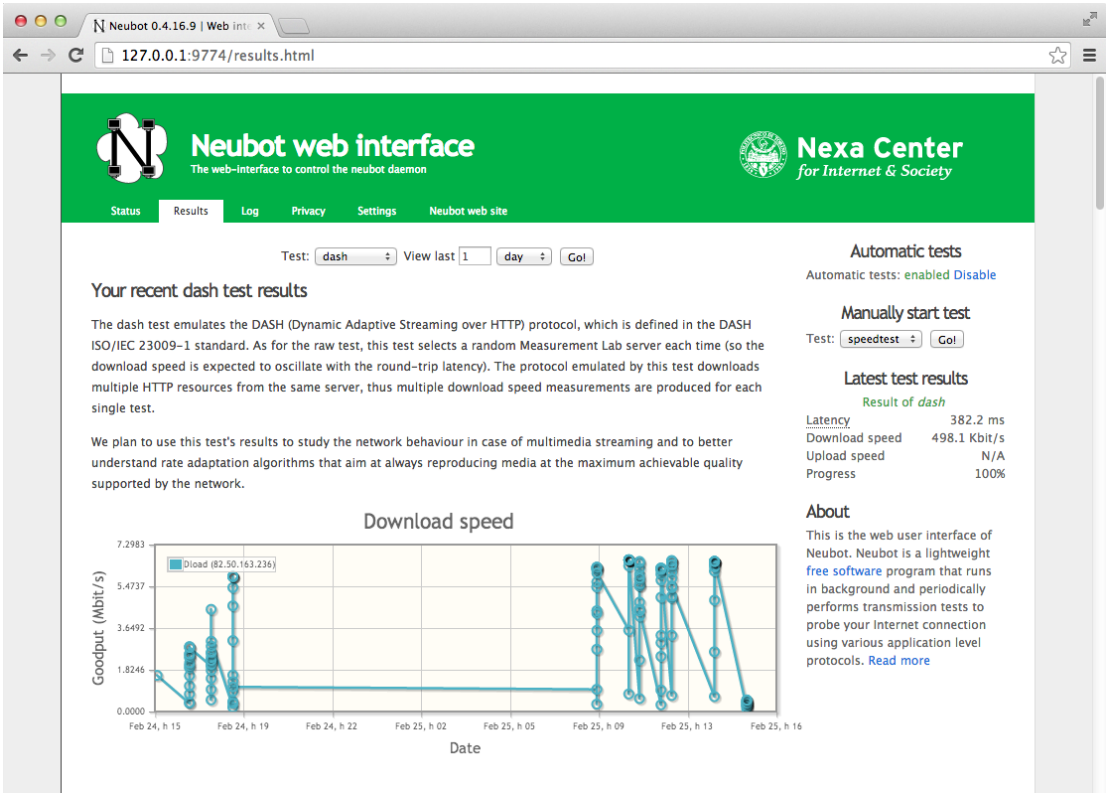


Figure 4.1: The Neubot Web interface.

Test Server, τ. Let us also assume that τ is the only Neubot Test Server. Now, if the TestController (i.e., the Test Server module that manages the queue) is restarted, all those instances will lose the connection and abort their test attempt. Accordingly, they will all schedule the next test to run after 25 minutes. Therefore, 25 minutes later 100 Neubot instances will connect to the Test Server to perform a test. This fact increases the Test Server load. Conversely, if δ is random in the 23–27 minutes interval, those instances will not connect with the Test Server precisely at the same time, thereby reducing the load on the Test Server and the average queue length seen by the average instance.

Having explained δ, we are now ready to describe the implemented tests, following the order in which the tests were implemented and added to Neubot:

The HTTP Speedtest

HTTP Speedtest is a HTTP-based test that downloads and uploads data from the closest-available Neubot Test Server using a single HTTP [211] connection. Originally the HTTP Speedtest test was inspired to the SpeedTest.net [136] broadband speed test developed by Ookla [120], hence the test name. The test measures the download and the upload speed at the application level. Moreover, the test estimates the round-trip time (RTT) using as a proxy the time that the connect system call takes to complete, as well as the time elapsed from sending a small HEAD request

to receiving the corresponding small response. The test transfers a number of bytes that guarantees that each phase of the test (download, upload) lasts for about five seconds (an empirically determined threshold which guarantees, in our experience, a good compromise between the test accuracy and the test duration). A more detailed description of this test is provided in Chapter 6.

The BitTorrent test

As also done by the HTTP Speedtest, the BitTorrent test measures the RTT, the download and the upload speed with the closest-available Test Server. The major difference between the BitTorrent and the HTTP-Speedtest tests is the following: the BitTorrent test uses the BitTorrent peer protocol [196] instead of the HTTP protocol. As for the Speedtest, the BitTorrent test transfers a number of bytes that guarantees that each phase of the test (download, upload) lasts for about five seconds (an empirically determined threshold which guarantees, in our experience, a good compromise between the test accuracy and the test duration). However, while Speedtest makes a single GET request for a large-enough amount of data, the BitTorrent test downloads many small chunks in a request-response fashion, to better emulate the BitTorrent protocol. As a consequence, to emulate the download of a large amount of data, the BitTorrent test needs to pipeline many requests at the beginning of the transfer. A more detailed description of this test is provided in Chapter 9.

The 'raw' test

The raw test performs a 'raw' 10-second TCP download to estimate the download goodput (the 10-second duration was chosen because other comparable tests, such as the NDT test [192], run for ten seconds). It is called 'raw' test because it directly uses TCP and it does not emulate any protocol. During the download, this test also collects statistics about the TCP sender using Web100 [251], which is installed on all M-Lab servers. In addition, it estimates the round-trip time in two ways: (1) by measuring the time that connect() takes to complete (like BitTorrent) and (2) by measuring the average time elapsed between sending a small request and receiving a small response. Unlike the HTTP Speedtest and the BitTorrent tests, the 'raw' test uses a random Neubot Test Server[2]. A more detailed description of this test is provided in Chapter 7.

The Dashtest test

The Dashtest test emulates the download of a video payload using the Dynamic Adaptive Streaming over HTTP (DASH) MPEG standard [227]. As the 'raw' test

[2]In this case, a random server is chosen to perform measurements in a wide range of conditions, since the 'raw test' was designed to support the study of models to estimate the packet loss rate from application level measurements. Conversely, the HTTP-Speedtest and the BitTorrent tests use the closest server because they aim to estimate the network performance.

does, Dashtest uses a random Neubot Test Server. The rate adaptation logic used by this test is the following. At the beginning of the test, the Dashtest client requests the first video segment using the lowest bitrate representation. During the download of the first segment, the client calculates the estimated available bandwidth of the downloaded segment by dividing the size of such segment (in kb) by the download time (in seconds). Afterwards, the Dashtest requests the next segment using the representation rate that is closer to the download speed of the current segment[3]. For example, if 200 kbit/s, 1 Mbit/s and 2 Mbit/s are the available bitrate representations and the estimated available bandwidth is 1.4 Mbit/s, the Dashtest will request the next segment using the 1 Mbit/s bitrate representation. The algorithm described above is, of course, used not only after the download of the first segment but also after the download of all the subsequent segments, thereby adapting the requested bitrate representation to the download speed. A more detailed description of this test is provided in Chapter 8.

Having presented the tests, let us describe the Neubot servers implementation.

4.3 Neubot-Servers Implementation

In this Section we describe how we implemented the Neubot Servers architecture described in Section 3.4. We start by discussing the master.neubot.org server.

master.neubot.org

The master.neubot.org server is deployed at the Torino-Piemonte Internet eXchange (TOP-IX) [141], the nearest Internet eXchange (IX) to the Politecnico di Torino. The virtual machine hosting master.neubot.org has the following specifications: 1-GByte RAM; 2 GHz Intel Xeon CPU; 1 core. The operating system is Debian 6. The virtual machine's Internet-access speed is 1 Gbit/s, both downstream and upstream.

Historically, the master.neubot.org server has been our unique server, on which we deployed all the Neubot Servers functionality. After Neubot was added to M-Lab (with Neubot 0.4.6 in January, 2012), the master.neubot.org server was less and less used as a Test Server. It still implements the Test Server functionality to provide support for very old clients and to cope with some special cases, however, nowadays virtually all the tests are performed by M-Lab servers. Moreover, since we implemented digitally-signed auto-updates for Win32 (with Neubot 0.4.14 in September, 2012), the master.neubot.org server is not used anymore to inform Neubot clients that their Neubot version needs to be updated. Currently, the master.neubot.org server is

[3]This strategy is the one used in the common case, i.e., when the available bandwidth does not change significantly during the test. Instead, when it estimates that the available bandwidth reduced, the test uses a more conservative strategy to pick the next-segment size (see Chapter 8).

only used to provide the Discovery Server functionality to the HTTP Speedtest and BitTorrent tests, which were implemented before M-Lab made available its mlab_ns service (on which we will return in a moment).

In the future, the master.neubot.org server will be the server on which we will deploy a Configuration Server capable of suggesting diverse testing policies to diverse Neubot instances, using as features to tailor the policy, among others, the ISP and the geographic location of the Neubot instance.

Test Servers

The current implementation of the Test Server runs the TestController and the TestProvider components (see Section 3.5) within the same system process. The Test Servers are hosted by M-Lab. In practice, M-Lab provides us with Linux-VServer [79] virtual-private servers (VPSs) in which a Neubot instance is run in Test-Server mode. As of October 2013, there are 124 M-Lab servers operating across 40 geographically-distributed sites around the globe. According to the technical requirements document (as of February, 2014) [86], M-Lab allocates twelve VPSs to a physical machine having the following, minimum requirements: bandwidth at least 1 Gbit/s; 8 cores; 8 GByte RAM; 100 GByte hard disk. Supported hardware are, for example, the Dell PowerEdge R620 and R610.

Discovery Servers

As mentioned above, for the HTTP Speedtest and BitTorrent tests we use a Discovery Server [111] installed at master.neubot.org. Such server uses GeoLite [11] to geolocate the connecting Neubot instances and, knowing which M-Lab servers are deployed in which country, it redirects a Neubot instance to the closest (in roughly geographical terms) server. Note that our Discovery Server does not know the status of an M-Lab server, therefore it may redirect a Neubot instance on a server that, e.g., may be down for maintenance. The logic [110] used by our Discovery Server to pick the closest Neubot instance is the following: when one or more servers are available for a country, we pick one of them at random; when no server is available for a country, we use a server of the same continent (given a country, we consistently pick the same server in the continent to avoid bouncing from servers with diverse latency); should no server be available in the same continent (a very unlikely condition, valid only for hypothetical users using Neubot from the Antarctica), we use master.neubot.org.

Conversely, the 'raw' and the Dashtest tests use a service developed by M-Lab and called mlab_ns [87]. Based on Google App Engine [153], mlab_ns knows all the available M-Lab servers, as well as which tools (e.g., Neubot) are *actually* running on which servers. To this end, mlab_ns uses Nagios [94]. Of course, this means that the Neubot Test Server must also implement a piece of software dedicated to responding to the probes sent by Nagios [112]. Mlab_ns has a Web API: clients send HTTP requests indicating the tool (e.g., Neubot) and the server-selection policy (closest-server or random-server). Mlab_ns responds to the query returning the IP

address and the domain name of a server, selected according to the user-specified policy, in which the requested tool (e.g., Neubot) is up and running. Of course, when the closest-server policy is requested, mlab_ns uses geolocation to infer the Neubot instance location, which is needed to select the closest server.

As a fallback, if the Neubot Discovery server and/or mlab_ns are down, the Neubot instance will select a Test Server at random [108]. This mechanism is – of course – suboptimal, but it allows a Neubot instance to perform tests even in cases in which the discovery mechanisms are out of order. Currently, the list of Test Servers known by an instance is compiled at release time, and never updated. As part of our future work we look forward to update the list of known servers using the Auto-Update server and to remember the recent closest-servers returned by mlab_ns, so that, in case of discovery-mechanisms outage, Neubot can use such information to select a closeby server (which is arguably better than selecting a random server).

Configuration Servers

As anticipated in Section 4.1, the Configuration Server functionality is not implemented yet. The instance, in fact, picks the next test to run at random: the 'raw' test is selected with 9% probability; the BitTorrent and the HTTP Speedtest tests with 27% probability each; the Dashtest with 36% probability [109]. The Neuviz project (described in Chapters 10 and 11), whose first prototype allows us to visualize the Neubot results, should be able – in the future – to automatically provide the information that a Configuration Server can use to suggest to a Neubot instance the optimal testing policy (in terms of test type and Test Server).

Collect Servers

On the Test-Server side, the result of a test is a JSON file containing the Test-Server-side and the instance-side measurements results (the instance-side measurements being provided by the Neubot instance during the Collect phase of the test, as described in Section 3.5). The Test-Server Backend module [102] writes such JSON file on the local filesystem in a directory that is exported via rsync [132]. In turn, the M-Lab data-collection pipeline (which was not designed by us but which happens to implement the Collect Servers functionality) every day pulls the JSON files from the Test Servers – using rsync – and publishes them on the Google Cloud Storage service [36] under a No-Rights-Reserved Creative-Commons-Zero 1.0 Universal waiver [30]. More specifically: on the Google-Cloud Storage the JSON files are aggregated in tarballs that contain the JSON files collected during a day by a specific tool (e.g., Neubot) running on a specific M-Lab server. For example, the tarball 20131008T000000Z-mlab1-lga01-neubot-0000.tgz contains the Neubot data saved on the mlab1-lga01 M-Lab server on October 8, 2013.

Neubot Results Database

In its basic form, the Neubot Results Database allows to download the entire set of Neubot results. Such functionality is implemented by the Google-Cloud-Storage

service [36] (of course not designed by us) from which one can download the tarballs containing Neubot data. In particular, the main ways in which one can download the Neubot data are explained by the M-Lab wiki [85]. In this vein, let us also mention data.neubot.org [103], our mirror of the Google-Cloud-Storage Neubot data. As regards more advanced functionality, we already mentioned the Neuviz [114] project (discussed in Chapters 10 and 11) which shall allow us to visualize the Neubot results; to spot anomalies; to improve our testing methodology; to help the Configuration Server to tailor the testing policy to a specific Neubot instance. Moreover, let us mention the the streaming.polito.it/neubot service (maintained by our colleagues and coauthors Antonio Servetti and Enrico Masala): such service, which we are planning to integrate with Neuviz, provides simple statistics like, e.g., the number of tests per day [133]. Moreover, in the future, Neubot results should be available through BigQuery, a Google service that allows one to run SQL-like queries on data saved on the Google-Cloud-Storage service [26]. BigQuery is another promising service that can be used in the future to implement some of the Neubot Results Database functionality.

Auto-Update Servers

The Auto-Update Servers functionality is implemented by the LAMP [156] releases.neubot.org server. On such server we host the tarballs containing the sources and the binaries of the Neubot releases, as well as the tarballs used to implement the automatic updates. As mentioned above, we digitally sign with the Neubot private RSA key all the items that one can download, allowing – in particular – a Neubot instance to verify the integrity and authenticity of the auto-update tarballs.

Having described the Neubot Servers implementation, let us now draw the conclusions.

4.4 Concluding Remarks

In this Chapter we described the current Neubot implementation. In doing that, we used as reference the Neubot architecture described in the previous Chapter. Written in Python and released under the GNU General Public License v3, Neubot runs in the background and currently implements four transmission tests, as well as automatic updates. The most relevant Neubot-architecture features and components that are not implemented yet are peer-to-peer tests and arbitrarily-long tests. To implement peer-to-peer tests we need to implement a mechanism to allow a Neubot instance to register as a Test Server and the NATTraversal module, which is crucial to allow a Neubot instance to bypass a NAT [285]. To implement arbitrarily long tests we need to implement the HostLoad module, of which a prototype exists for Linux only. Other unimplemented features (e.g., a version of Neubot for mobile phones and the dynamic testing policy) are currently under development.

Afterwards, we briefly described the four implemented tests. The HTTP Speedtest is based on the HTTP [211] protocol and measures the RTT, the download and the upload speeds. The BitTorrent test is based on the BitTorrent peer protocol [196] and measures

the RTT, the download and the upload speeds. Both tests perform measurements with the closest available server only. The 'raw' test – which instead performs measurements with a random server – does not emulate any protocol and works on top of TCP (hence the name). It measures the download speed and the RTT, and uses Web100 [251] on the server to measure also a number of TCP variables. The Dashtest emulates a Dynamic-Adaptive-Streaming-over-HTTP (DASH) [227] client: it downloads emulated video segments selecting the representation rate that is closer to the download speed of the previous segment. Also the Dashtest runs measurements with a random server. We will return to the four Neubot tests in the following Chapters of this thesis.

Subsequently, we described the Neubot servers. The Test Servers (more than 100) are provided by M-Lab, a consortium that hosts the test servers of open network-performance measurement tools such as Neubot. On M-Lab servers we install an instance of Neubot that is configured to run in Test-Server mode. Such instance implements the TestController and the TestProvider within the same process. The master.neubot.org server – deployed in Turin and hosted by the TOrino-Piemonte Internet eXchange (TOP-IX) [141] – is used to implement the Discovery Server functionality of the HTTP Speedtest and BitTorrent tests. Conversely, we use the mlab_ns service provided by M-Lab to implement the Discovery Server functionality for the 'raw' and the Dashtest tests. The Configuration Server is currently not implemented: at the moment each Neubot instance selects the next test to run with fixed probability. The Collect Servers functionality is implemented by M-Lab, which periodically fetches the data from the Test Servers using rsync [132]. The Neubot Results Database functionality is again implemented by M-Lab, which automatically publishes the Neubot results on the Google Cloud Storage [36] every day. We also have a mirror of the Google-Cloud-Storage data: on top of such mirror we are building Neuviz (see Chapter 10), which should help us to analyze the Neubot results and which should help us to implement a dynamic Configuration Server. The Auto-Update Servers functionality is implemented by the releases.neubot.org server, in which we publish new releases and digitally-signed automatic-updates.

As regards future work, the development of Neubot is proceeding in the following directions. We are working to port Neubot to Android [23]. To do so, we are writing an Android application based on a Neubot library partially written in C/C++ (as discussed in Section 3.6) and based on Libevent [75]. We have not decided yet which high-level language is more adequate for the remainder of the Neubot library: our current candidates are Lua [80] and Python [130]. We are also working on a new network-performance test – possibly based on Libutp [77] – for the Micro Transport Protocol (uTP) [259]. We are also working to improve the plugins architecture, measure the host load, and implement peer-to-peer tests. In addition there is also the Neuviz effort mentioned above and more extensively described in Chapter 10.

Chapter 5

The Neubot Project History and Numbers

In this short Chapter we list the most relevant events in the Neubot history and we provide some Neubot numbers, to provide the reader with an idea of the dimension of Neubot as a software project.

5.1 Neubot History

We list the most relevant events in the Neubot history divided in releases, papers, and other events.

Releases

- **2008-06** – First Neubot prototype, written by Gianluigi Pignatari Ardila in the context of his M.Sc. thesis "Design and Implementation of a Distributed System to evaluate Net Neutrality", under the supervision of prof. Juan Carlos De Martin;
- **2010-03-15** – Simone Basso joins the Nexa Center for Internet & Society[1] as a research fellow, to continue the development of Neubot;
- **2010-11-03** – First public release; after nine months of development, Neubot 0.3.0 is released for Windows and Debian, featuring the Speedtest test described in Section 6.2;
- **2011-02-25** – Neubot 0.3.5 released for Windows, Debian, and MacOS: includes an improved Speedtest test (as described in Section 6.4) and the Neubot Web user interface described in Chapter 4;

[1] An independent multidisciplinary research center, hosted by the Dept. of Computer and Control Engineering at Politecnico di Torino. The center is co-directed by prof. Juan Carlos De Martin and prof. Marco Ricolfi, and it has been sponsoring the development of Neubot since 2008.

- **2011-05-20** – Neubot 0.3.7 released for Windows, Debian, and MacOS: includes a rewritten Speedtest test (see again Section 6.4) and the first version of the privacy policy (not described in this thesis);

- **2011-07-20** – Neubot 0.4 released for Windows, Debian and MacOS: adds the BitTorrent test (described in Chapter 9);

- **2011-09-19** – Neubot 0.4.2 released for Windows, Debian and MacOS: adds automatic updates for MacOS, improves the Speedtest test as described in Section 6.7, preparatory work to integrate Neubot and M-Lab;

- **2011-10-25** – Neubot 0.4.4 released for Windows, Debian, MacOS, and FreeBSD: Neubot was added to FreeBSD, other refinements;

- **2012-01-24** – Neubot 0.4.6 released for Windows, Debian, MacOS, and FreeBSD: finished to integrate Neubot with M-Lab, new privacy policy (see Section 4.1) by which both the permission to collect and the permission to publish the IP address must be given (so that all Neubots can use M-Lab: the open-data policy of M-Lab, in fact, requires that the test results are openly published on the web, including the IP address);

- **2012-09-30** – Neubot 0.4.14 released for Windows, Debian, MacOS, and FreeBSD: add automatic updates for Windows;

- **2012-10-14** – Neubot 0.4.15.5 released for Windows, Debian, MacOS, and FreeBSD: add the 'raw' test (see Section 7.6), add support for mlab_ns (the M-Lab name server, see Section 4.3), start using Web100 for the 'raw' test;

- **2013-10-30** – Neubot 0.4.16.9 released for Windows, Debian, MacOS, and FreeBSD: add the Dashtest (see Section 8.2), add support for tests as plugins (see Section 4.1), add mini-language to simplify the generation of the results (see Section 4.1).

Papers

- **2008-06-27** – Paper "The Neubot Project: A Collaborative Approach To Measuring Internet Neutrality" [201] by Juan Carlos De Martin and Andrea Glorioso, presented by Juan Carlos De Martin at the IEEE Symposium on Technology and Society (ISTAS 2008). This paper was mainly used to prepare Chapter 3 of this thesis;

- **2010-10-01** – Paper "Rationale, Design, and Implementation of the Network Neutrality Bot" by Simone Basso, Antonio Servetti and Juan Carlos De Martin, presented by Simone Basso at the Congresso Nazionale AICA 2010. This paper was mainly used to prepare Chapter 3 of this thesis;

- **2011-07-01** – Paper "The network neutrality bot architecture: a preliminary approach for self-monitoring of Internet access QoS" [176] by Simone Basso, Antonio Servetti and Juan Carlos De Martin, presented by Simone Basso at the IEEE 16th International Symposium on Computers and Communications (ISCC'11). The paper won the best-student paper award. This paper was mainly used to prepare Chapters 3 and 6 of this thesis;

- **2011-11-16** – Paper "The hitchhiker's guide to the Network Neutrality Bot test methodology" [175] by Simone Basso, Antonio Servetti, and Juan Carlos De Martin, presented by Simone Basso at the Congresso Nazionale AICA 2011. This paper was mainly used to prepare Chapter 6 of this thesis;

- **2012-08-17** – Paper "Estimating packet loss rate in the access through application level measurements" [177] by Simone Basso, Michela Meo, Antonio Servetti, and Juan Carlos De Martin, presented by Simone Basso at the ACM SIGCOMM Workshop on Measurements Up and Down the Stack (W-MUST) 2012 hosted by the ACM SIGCOMM Conference. This paper was mainly used to prepare Chapter 7 of this thesis;

- **2013-07-01** – Paper "Strengthening measurements from the edges: application-level packet loss rate estimation" [178] by Simone Basso, Michela Meo, and Juan Carlos De Martin, published on the July 2013 issue of the ACM SIGCOMM Computer Communication Review. This paper was mainly used to prepare Chapter 7 of this thesis;

- **2013-09-01** – Paper "Challenges and Issues on Collecting and Analyzing Large Volumes of Network Data Measurements" [245] by Enrico Masala, Antonio Servetti, Simone Basso, and Juan Carlos De Martin, presented by Antonio Servetti in the context of the ADBIS Special session on Big Data: New Trends and Applications (BiDaTA) 2013 hosted by the 17th East-European Conference on Advances in Databases and Information Systems (ADBIS) 2013. This paper was mainly used to prepare Chapter 10 of this thesis;

- **2013-09-20** – Paper "Visualizing Internet-Measurements Data for Research Purposes: the NeuViz Data Visualization Tool" [217] by Giuseppe Futia, Enrico Zimuel, Simone Basso, and Juan Carlos De Martin, presented by Giuseppe Futia in the context of the Congresso Nazionale AICA 2013. This paper was mainly used to prepare Chapters 10 and 11 of this thesis;

- **2013-12-08** – Notification of acceptance of the paper "The NeuViz Data Visualization Tool for Visualizing Internet-Measurements" [218], by Giuseppe Futia, Enrico Zimuel, Simone Basso, Juan Carlos De Martin, which will be published on the February-2014 issue of the Mondo Digitale journal. This paper was mainly used to prepare Chapters 10 and 11 of this thesis;

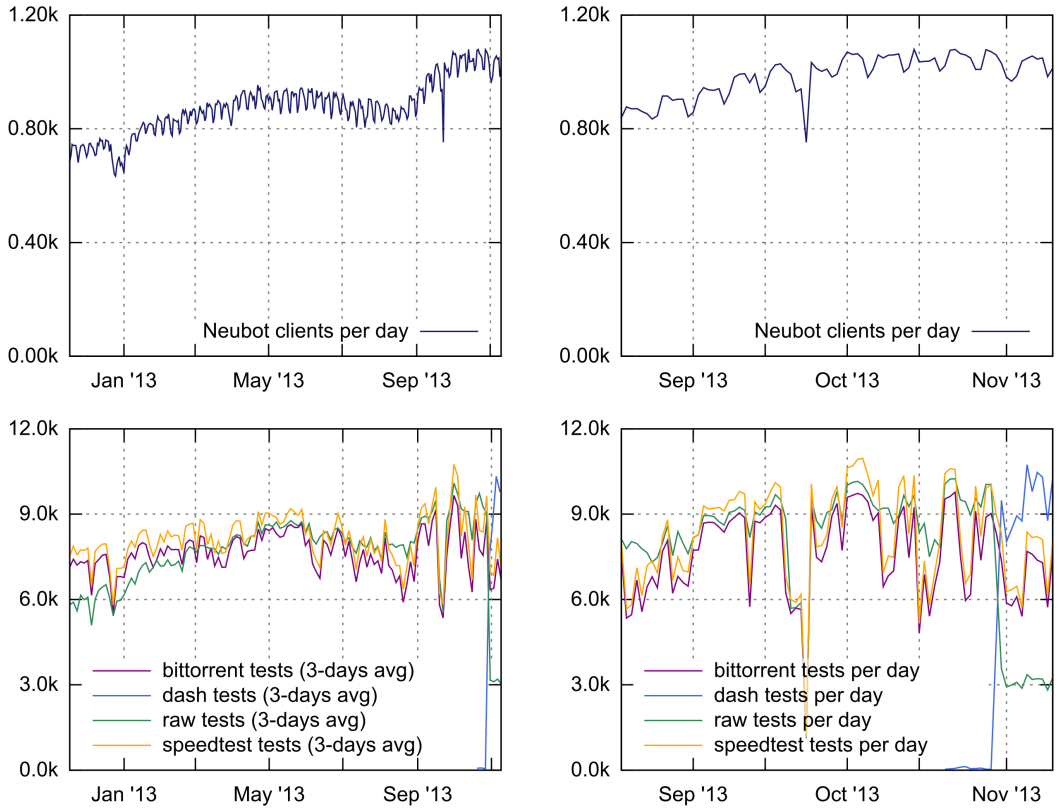


Figure 5.1: Number of clients and the number of tests per day in January–November, 2013 (left) and in September–November, 2013 (right).

- **2013-12-24** – Notification of acceptance of the paper "Measuring DASH Streaming Performance from the End Users Perspective using Neubot" [179], by Simone Basso, Antonio Servetti, Enrico Masala, Juan Carlos De Martin, which will be presented at the ACM Multimedia Systems DataSet Session 2014. This paper was mainly used to prepare Chapter 8 of this thesis.

Other Events

- **2011-06-27** – Attended the closed-door workshop organized the BEREC (Body of European Regulators for Electronic Communications) in Brussels, in which Simone Basso and professor J.C. De Martin presented Neubot;
- **2011-09-29** – Published the first batch of Neubot data under a Creative Commons Zero waiver [30] at `http://www.neubot.org/data`;
- **2011-11-02** – Neubot presented at the Open World Forum in the panel "A tour of Network Neutrality Solutions", moderated by Jeremie Zimmerman (La Quadrature Du Net), with John Palfrey (Berkman Center for Internet & Society at Harvard University), Daphne van der Kroft (Bits of Freedom) and Simone Basso;

- **2011-11-09** – Simone Basso attended the 2011 EU Hackhathon and worked with the M-Lab staff (especially Thomas Gideon and Tiziana Refice) to move forward the integration of Neubot and M-Lab;
- **2011-12-12** – Neubot starts using M-Lab, but only for users that provided both the permission to collect and the permission to publish their IP address (as required by the M-Lab open-data policy);
- **2013-03-15** – Awarded an unrestricted $52,500 research grant by M-Lab/Google to continue the research work on Neubot;
- **2013-12-02** – Started the development of Neubot for Android (see Section 3.4.2);
- **2014-02-11** – Contacted by Richard Jones, directory of Espy Consulting, to adapt Neubot to "support a local government project in England that is funding a major upgrade to the Broadband Infrastructure." We provided help. Espy Consulting submitted the project to the local government and is currently waiting for a response.

5.2 Neubot Numbers

Neubot was downloaded over 5,000 times in the last year (data updated to mid November, 2013). On the average, in the September–November 2013 period there were 1,000 active users per day, 27,000 tests per day (2,374,870 in total), and 300 Mbyte of collected data per day. Moreover, in the same period, Neubot was used by 2,067 unique users and by 29,275 unique IP addresses, which - according to the GeoIP database [11] - can be mapped onto 107 countries, 1,034 autonomous systems, and 4,606 diverse geographic locations. In the same period, the countries with more tests were: the U.S. (980,211; 41%), Italy (241,202; 10%), the U.K. (190,369; 8.0%), Germany (145,285; 6.1%), Canada (126,923; 5.3%), France (116,541; 4.9%), Australia (88,000; 3.7%), Brazil (426,71; 1.8%), and Austria (33040; 1.4%). These countries account for more than 80% of all the tests run by Neubot.

Figure 5.1 shows the number of clients and the number of tests per day in the last year and in the September–November 2013 period.

As of November, 2013, Neubot contains more than 32,000 lines of Python, more than 3,000 lines of JavaScript, 657 lines of HTML, and more than 1,000 lines of bourne shell scripts. Overall, nine developers (eight volunteers plus Simone Basso) contributed to the project more than 4,000 patches.

Chapter 6

Experiment #1: HTTP Speedtest

The protagonist of this Chapter is the first experiment developed for Neubot, the 'HTTP Speedtest' (or, simply, 'Speedtest'). Called Speedtest because it was originally inspired to the Speedtest.net broadband-speed test, the HTTP Speedtest test was crucial to Neubot because it showed that the Neubot architecture fitted the purpose of collecting, from the network edges, network-performance data on a large scale. With subsequent methodology refinements and experiments, we also showed that the Speedtest results could be used to benchmark the performance of the Internet-service providers (ISPs) serving a given geographic area. In addition, our studies to improve the Speedtest methodology eventually enhanced our understanding of broadband-speed and network-quality measurements and refined our research questions, which were further developed as described in the following Chapter.

This Chapter is organized as follows. In Section 6.1, we explain what motivated us to write the Speedtest test. In Section 6.2, we describe the initial implementation of the Speedtest. Afterwards, in Section 6.3, we describe results that allowed us to conclude that the Neubot architecture fitted the purpose of running network experiments from the network edges, drawing from our best-student-paper-awarded ISCC 2011 work [176]. Successively, in Section 6.4, we describe a first round of enhancements to the Speedtest to increase its accuracy. In Section 6.5 we provide a qualitative analysis of the performance of several widely-used ISPs in the Turin area, drawing from our AICA 2011 work [175]. Afterwards, in Section 6.6, we discuss the Speedtest methodology, including the comments received during the review process that preceded the admission to the M-Lab platform. In Section 6.7, we describe the Speedtest enhancements implemented after being accepted in M-Lab. Finally, we draw the conclusions.

6.1 Motivation

The Speedtest test was the first test implemented for Neubot. In fact, this test was already part of Neubot before the first public release (Neubot 0.3.0 released on November 2, 2010). The initial objective of the Speedtest was to estimate the download and upload speeds of an

access network, as well as the round-trip time (RTT) latency between the Neubot instance and the measurement server. Our work was heavily inspired to the namesake test available at Speedtest.net and designed by Ookla [136]. Later, the test methodology changed, but the Python file name remained 'speedtest.py' and, in turn, the test itself was eventually called 'Speedtest'. To be fair, it was not our brightest idea to call such test Speedtest, because it could be easily confounded with the more famous Speedtest by Ookla. To avoid any confusion, therefore, in the following we will call the more famous cousin either 'Ookla Speedtest' or 'Speedtest.net'.

We added the Speedtest to Neubot because we needed an initial network-performance experiment to validate the Neubot architecture and its initial implementation. To this end, it seemed reasonable to implement a test that tried to measure the broadband speed of a user connection: in fact, because many other sources were already providing broadband speed measurements [136] [192] [15], our reasoning was that we could have used the results of such experiments as a yardstick for our own experiments. To be fair, we were not totally aware, at the time, of the methodological differences of the above-cited sources: for us they were all trying to 'measure broadband speed'; we will be more precise on what it means to measure broadband speed in Section 6.4.

At the time (November 2010 – January 2011) our main source of knowledge on broadband speed measurements was the NetForecast report [115] on the broadband speed measurement technique used by comScore [39]. According to the executive summary of such report, the comScore methodology was not accurate enough (a) because it used a single TCP connection, which did not allow to test a wide range of bandwidth-delay products[1], and (b) because there were cases in which the delay between the client and the server was too high, thereby reducing the measured speed[2]. Therefore, we wrote a simple HTTP test that used two connections (to increase the range of tested bandwidth-delay products) and, because we had a single server at the time (provided by TOP-IX [141], the TOrino-Piemonte Internet eXchange, and located in Turin), we considered only the results of closeby (in terms of RTT) experiments.

Successively, following a learning-by-doing approach: we released the first public version of Neubot (0.3.0), we started to study the results of our first implementation of the Speedtest and, in parallel, we started to study the literature, to understand the results and the limits of the Speedtest, as well as how to make it more accurate. An account of what we learned afterwards on Neubot, on the Speedtest methodology and on our research questions is given in the following Sections of this Chapter. We start by describing more

[1]The bandwidth-delay product represents the maximum number of bytes that an end-to-end path can hold. Because TCP uses a window protocol to decide how much data can be injected into the network at a given time, and because the window may not grow beyond a certain threshold, if the bandwidth-delay product, β, is larger than the maximum window, μ, the performance is limited by μ rather than by β. To avoid this problem, one can use α parallel connections such that $\alpha \cdot \mu > \beta$.

[2]According to the well-known Mathis model [250] the performance of a steady-state TCP connection is inversely proportional to the RTT and to the square root of the packet loss rate (PLR). For more information on the Mathis model, see Chapter 7.

in detail how the first version of the Speedtest was implemented.

6.2 The Initial Speedtest: Neubot 0.3.0–0.3.4

Here we describe the initial Speedtest implementation. This description is based on the 2011 ISCC paper [176], only more abstract, in order to provide the framework that we will use throughout this Chapter to describe subsequent methodology changes.

Let us start from the basics. The Speedtest takes place between a Neubot instance – henceforth, 'the instance' – and a test server (when M-Lab is used, the server is always the closest test server). Before the actual test, there is the negotiation phase: test parameters are negotiated and the test itself may be delayed to control the server load. Then the actual test, discussed below, takes place. After the test, there is the collect phase: the instance sends the results to the test server and *vice versa*. For a more detailed explanation of the test phases (negotiation of the test parameters, test itself, data collection), we refer the reader to Chapter 3.

More specifically, the Speedtest uses the HTTP protocol and measures the round-trip time (RTT), the download goodput and the upload goodput. The round-trip time is estimated using two distinct techniques: the measurement of the time required to connect and the measurement of the request-response latency. The goodput is the application-level throughput [168], calculated dividing the amount of sent (or received) bytes by the elapsed upload (or download) time. We warn the reader that sometimes, in the following, we will use interchangeably the words goodput and speed; however, whenever the (less precise) word speed is used, it is intended that we mean (the more precise) goodput, unless it is otherwise specified.

In the following paragraphs we provide more details regarding the measurements, starting from the two ways in which the RTT is measured.

Time Required to Connect The time required to connect (also called, in the following, 'connect delay') is the time that the connect() system call takes to complete. It approximates the round-trip time because the connect() system call immediately sends a SYN segment and returns as soon as it receives the corresponding SYN|ACK segment. Regarding the implementation, Neubot uses nonblocking I/O and select() to dispatch I/O readiness events; therefore, in practice the connect time, τ, is the difference between the time when the writable socket is confirmed to be actually connected [183] and the time when connect() returns the EINPROGRESS socket-level error. The τ interval is by construction greater than the RTT, since the network latency is, typically, significantly greater than the TCP/IP stack latency, therefore, we can safely use τ to approximate the RTT.

Moreover, let us remember that the Speedtest methodology was revised a number of times since it was added to Neubot. The current implementation, in particular, uses a single TCP connection, but previous implementations used up to four TCP connections (we decided to use a single TCP connection to more accurately measure the broadband quality, as explained in Section 6.6). In such implementations, the connect delay reported

by the Speedtest test was the average connect delay measured by the up-to-four connections. Moreover, it should be noted that before Neubot 0.3.7 (released on May 20, 2011) concurrent connection attempts were not serialized, leading to biased connect-time measurements.

Request-Response Latency The request-response latency is computed performing the difference between the time when an HTTP response is received and the time when the corresponding request was sent. This implementation guarantees that the measured time delay is a reasonable overestimation of the RTT: in fact, Neubot sends small HEAD requests and, following RFC2616-requirements for responding to HEAD requests [211], the server responds with very small, bodyless responses that can fit into a single Ethernet frame. To measure the request-response latency more accurately, the client repeats the measurement for ten times, and the result is the average of the ten measures. Also, let us remark again that the Speedtest test underwent a number of methodology revisions, which lead it to use up to four TCP connections. Currently the test uses just one connection and this situation has now been established for a long time (since September, 2011). Anyway, even when the number of connections was greater than one, the test used only one connection to measure the request-response latency.

Download Speed The goodput is calculated dividing the amount of received bytes by the elapsed time, computed as the difference between the time when the last byte of the response is received and the time when the first byte of the request was sent. The amount of bytes requested using GET is initially small, to avoid overloading slow networks. However, if the elapsed time is lower than the target elapsed time (three seconds in the current implementation), the test is repeated, increasing the amount of bytes requested so that, under constant network conditions (i.e., bandwidth and RTT), the repeated-test elapsed time should be greater or equal than than the target elapsed time. For example, if the target elapsed time is three seconds, we request 100 bytes, and the elapsed time is one second, we repeat the test asking for 300 bytes.

More precisely, Algorithm 1 (see next page) describes the generalized Speedtest download-only class using a pseudo-Python notation. It is composed of two methods, *init* and *run*: the former is called only once, to setup the download parameters; the latter is called each time a new Speedtest is run. The *init* parameters are the number of concurrent connections, ν, and the size of the socket receive buffer, σ. As regards the *run* method, the *http_get* function uses $nflows$ concurrent TCP connections to download *nbytes* (with each connection) using the GET method. The *http_get* function returns the download time, δ, which, when more than one connection is used, is the average of the download times. The $sobuf$ parameter is treated as follows: if $sobuf$ is zero, the Speedtest lets the operating system manage the receive buffer (some systems scale the buffer automatically to maximize the performance [213], other systems use a fixed hopefully-large-enough buffer); otherwise, the buffer value is forced to be equal to $sobuf$. The *adapt_nbytes* and *was_too_short* functions takes care of, respectively, (i) updating the number of bytes to be downloaded

Algorithm 1 Generalized Speedtest download-only class

Require: $\sigma \geq 0, \nu > 0$
  class SPEEDTEST
    method INIT(σ, ν)
      $nbytes \leftarrow 64000$
      $sobuf \leftarrow \sigma$
      $nflows \leftarrow \nu$
    method RUN()
      $\eta \leftarrow 1$
      while True
        $\delta \leftarrow http\_get(nflows, nbytes, sobuf)$
        $nbytes \leftarrow adapt\_nbytes(nbytes, \delta)$
        $\eta \leftarrow \eta + 1$
        if not $was\_too\_short(\eta, \delta)$
          break

next and (ii) deciding whether the download test has run for long enough. As the reader can see, Algorithm 1 does not show the implementation of these two functions, because they changed significantly over the lifetime of the Speedtest test. We can describe, in fact, all the methodology changes that we made to the Speedtest algorithm using different values of ν and σ as well as different implementations of the *adapt_nbytes* and *was_too_short* functions.

Before we continue the description, let us underline the following (perhaps obvious) fact: because *nbytes* is updated after each download, typically only the first download requires more than one iteration. In fact, subsequent tests start with a value of *nbytes* that should guarantee a long-enough download (unless the network conditions radically change).

The initial Speedtest implementation used $\sigma = 0$, $\nu = 2$. We used $\sigma = 0$ simply because we were not aware of the possibility (as well as of the advantages and the disadvantages) of forcing the socket receiver-buffer value. As regards the choice to use $\nu = 2$, it was motivated by the desire to increase the range of served bandwidth-delay products (BDPs): at the time, in fact, we only had a single server, master.neubot.org, located in Turin and provided by the TOrino-Piemonte Internet eXchange (TOP-IX).

Moreover, in the initial implementation *was_too_short* returned true when δ was lower than one second and η was lower than four. The former check was added because we wanted to perform a quick measurement: Neubot is a background tool that shall not consume too many resources and using the bandwidth for just one second has a very limited impact on the foreground user activities. The latter check (η lower than four) was added because the server could not return more than a fixed amount of bytes, μ; therefore, the test could possibly enter into an infinite loop if the requested *nbytes* was larger than μ.

Moreover, in the initial implementation *adapt_nbytes* returned $2 \cdot nbytes$, if δ was lower than one; otherwise *nbytes* was returned. We chose an exponential increase to quickly grow the amount of transferred bytes and find (within the first four iterations of the test) an

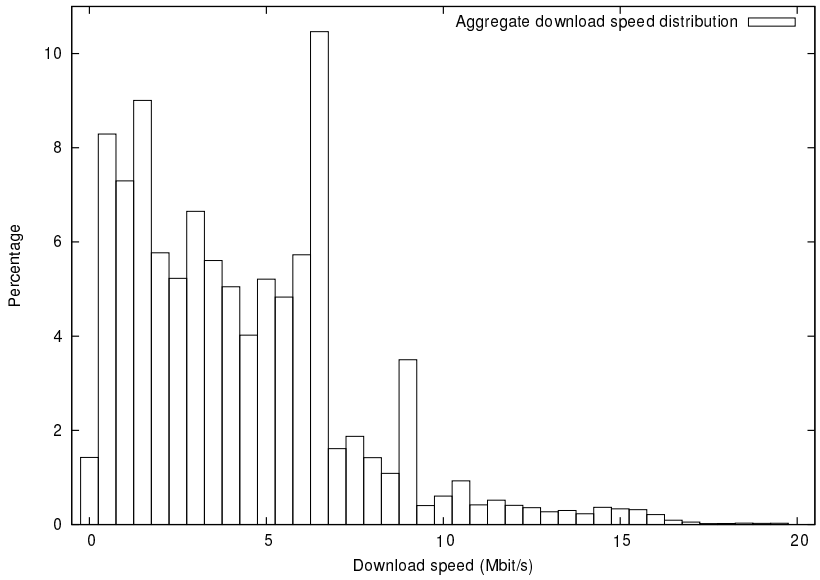


Figure 6.1: Aggregate download speed distribution (448,317 tests from October 3, 2010 to January 17, 2011, performed by 1,035 Neubot instances located near Turin and using as server master.neubot.org located in Turin).

amount of bytes large enough to appreciably load the network.

Upload Speed The upload speed measurement used the same methodology used for the download speed, except that: (a) the speed was measured at the sender, rather than at the receiver; (b) the POST method was used, instead than GET; (c) the content sent to the server was a a portion of the data downloaded during the download test (the assumption being that the upstream speed is smaller than the downstream speed). Upload speed measurements always mirrored download speed measurements, in terms of choices of the parameters of Algorithm 1. Since the upload-speed implementation is identical to the download implementation except for the three points (a)-(c) above, in the following, for brevity, we will always refer to the download implementation.

6.3 Evaluation of the Neubot Architecture

In this Section we evaluate the Neubot architecture to assess its fitness for the purpose of running collaborative network experiments from the network edges. To this end, we study the preliminary results of the Speedtest test, extracted from our database that contains tests from October 3, 2010 to January 17, 2011 for a total of 448,317 tests performed by 1,115 unique Neubot instances (henceforth, 'instances'). The involved Neubot versions are 0.3.0–0.3.3, therefore the testing methodology was always the one described in the previous Section. We focus on the download speed because it is a performance metric commonly used in ISPs advertisements. In particular, because the typical advertisement says "up to"

a given speed, we are interested to study the difference between the advertised speed and the measured speed when downloading from a well-known nearby network server (i.e., the master.neubot.org server provided by TOP-IX). To do so, we proceed as follows: we identify a reasonable 'goodset' of the collected results, which contains (as we shall see) only tests performed by Neubot instances attached to broadband networks close to our server; then, operating on the goodset, we show the distribution of the download speed, we compute the average speed, and we compare such average with the averages reported by Speedtest.net [136] and YouTube [15]. Afterwards, we select a Neubot instance within the goodset and we show the distribution of the download speed measured by that specific instance.

More specifically, the goodset is defined as follows: we consider only tests in which the measured download speed was lower than 20 Mb/s and the estimated RTT[3] was lower than 100 ms. Overall, the goodset includes 338,615 tests, i.e., 75% of the total number of tests; moreover, the tests in the goodset were performed by 1035 distinct Neubot instances (94% of the total). The restriction on the download speed is to single out tests performed from ADSL connections. Our database contains, in fact, a number of 100 Mb/s results originating from the Politecnico di Torino's campus or from TOP-IX, but the best-rated download speed for consumer ADSL in Italy was 20 Mb/s at the time; therefore, we chose 20 Mb/s as a cut-off value. The restriction on the RTT latency removes results collected from locations 'too far away' from our server, to avoid considering, as discussed in Section 6.1, the cases in which the speed was too much influenced by high latency.

Aggregate Download Speed Distribution The plot in Figure 6.1 shows the download speed distribution computed on the instances belonging to the above-defined goodset. From the analysis of this data, we clearly see peaks before 1 Mb/s, 2 Mb/s, and 7 Mb/s, which are three common average ADSL download speeds in Italy.

Average Download Speed Distribution We calculated the average download speed distribution in the goodset, and we compared such average with the one provided by Speedtest.net [136] and YouTube [15]. We do not show the variance because that makes little sense for results collected from different connections with different access speeds. We compare the goodset average with Speedtest.net's [136] 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 [15], 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 6.1: the averages are quite close. In particular, our average is very close to the Speedtest.net average, and it is greater than the YouTube average. We speculate that our average is comparable to the Speedtest.net average and greater

[3]To do so, we used the request-response latency only, because the connect time was buggy in these versions of Neubot, since the DNS resolve time was mistakenly measured as well.

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

Table 6.1: Average results of Neubot compared to the average results of Speedtest.net in Italy and YouTube Video-Download History in Turin.

than the YouTube average because our methodology is more similar to the Speedtest.net methodology than it is to the methodology of YouTube. If fact, we use two connections, just like Speedtest.net, while YouTube only uses a single connection: as we will discuss later in this Chapter, compared to a single connection, multiple connections use the bandwidth better and yield a higher goodput in presence of moderate congestion.

Single-Neubot Download Speed Distribution In this subsection we comment on the distribution of the download speed of a specific Neubot instance. The selected instance's tests consistently fall within the goodset. It has performed 1,383 tests and the RTT latency 84% of the time is smaller than 100 ms. Moreover, it appears the be attached to a 7 Mb/s ADSL connection. Indeed, a simple whois lookup shows that the IP addresses employed by such instance belong to an ISP whose top offer features 7 Mb/s downstream. Furthermore, the instance's download speed is always lower than 7 Mb/s.

Figure 6.2 shows the distribution of the download speeds measured by the selected instance. Note that we extracted the results of this instance from the database, given that each instance is identified by a random unique identifier. Since the results are also saved on the local disk of each instance, the owner of the selected instance could have done the same analysis, accessing the local database and using custom scripts or via the Neubot instance's web interface. Note, in particular, that most of the results in Figure 6.2 are distributed in proximity of 7 Mb/s, but there are significant spikes near 4 Mb/s and 1 Mb/s too (two other common downstream speeds for ADSL in Italy). As we shall see in the following Sections, these spikes were artifacts that could be avoided by either using more TCP connections or by running longer Speedtest tests.

Concluding Remarks This Section has shown that Neubot can be successfully used to crowdsource network experiments from the edges. The more than 1,000 users that installed Neubot, in fact, contributed to run a network experiment that, albeit in its infancy, allowed us to measure the typical broadband speed of instances geographically close to our unique server with results comparable to the ones provided by Speedtest.net [136] and YouTube [15]. In the next Section we will describe enhancements of the testing methodology and we will provide a first, tentative explanation of the the low-speed spikes observed in Figure 6.2 (an explanation that will later be refined in Section 6.6).

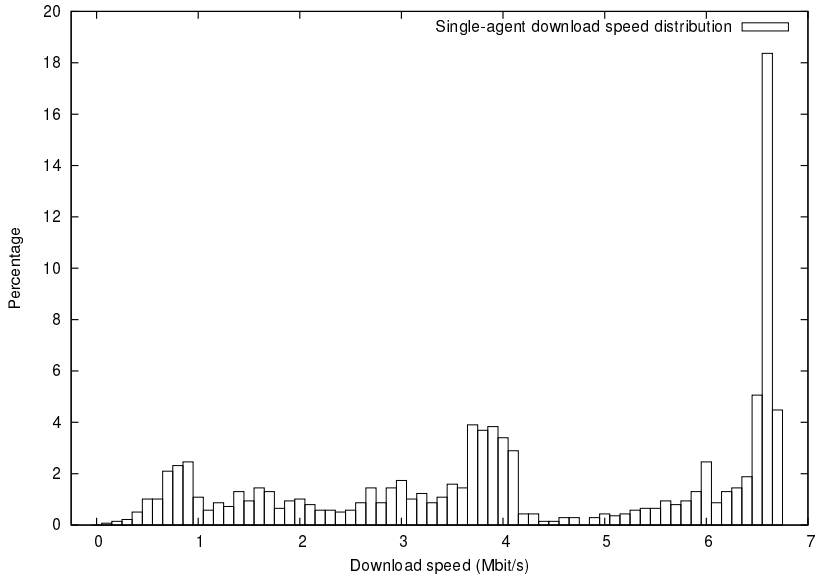


Figure 6.2: The download speed distribution for a specific Neubot instance (the mode value of the distribution is 6.7 Mb/s).

6.4 Evolution of the Speedtest: Neubot 0.3.4–0.4.1

Just after the submission, at the beginning of 2010, of the ISCC paper [176] (whose results were described in the previous Section), we turned our attention on the Speedtest again. The objective was to understand the low-speed spikes observed in Figure 6.2. Moreover, we were contacted by a user who had compared the result of our Speedtest with Ookla Speedtest: he/she asked us why our test reported a speed of 1 Mb/s while Ookla Speedtest reported 4 Mb/s, when the nominal downstream speed was 5 Mb/s. We asked the user to run the *mtr* network tool [93], and we discovered that such user was attached to an ISP in which there was significant congestion in the backhaul: the speed-limiting factor, in fact, was an overloaded router in the backhaul rather than the access network. Therefore, the Ookla Speedtest reported a result closer to the nominal downstream speed because it was using more connections (the Ookla Speedtest can use up to eight parallel connections, when we used just two single connections) and because it was running for longer time (about twenty seconds vs. one second). We studied thoroughly the problem and we ran experiments; we concluded that a first reasonable attempt to improve our Speedtest was to raise the number of parallel connections from two to four (another option being to run longer tests). We implemented this change in Neubot 0.3.5.

Neubot 0.3.5 In Neubot 0.3.5 (released on February 24, 2011) the Speedtest was modified to use $\nu = 4$, following the well-known recommendation to use four connections to increase the bandwidth efficiency in moderately congested wide area networks (WANs) [295]. Four parallel connections, in fact, deliver a fair goodput in congested networks because they are aggressive: when a connection is slowed down by the losses, the other three increase their

bandwidth usage. We asked the user who reported the problem to test again his access network with four connections, and the result was more satisfactory than before. Predictably, there were less low-speed spikes in the new-Speedtest data than in the old-Speedtest data, since four connections indeed used the bandwidth more efficiently.

Digression: Broadband Speed Measurements There is a tension embedded in the concept of measuring the broadband performance (and particularly the speed). Our user was clearly attached to a "bad quality" ISP (there were high losses in the backhaul), therefore, was it correct that, with the new Neubot-0.3.5 Speedtest methodology, the user's ISP appeared "better" (i.e., faster) than before?

The response is that, at the time, our goal was not clear, as we were trying to measure both the broadband speed (of the last mile) and the (overall) ISP quality at the same time, which is hard to do using the same network experiment. As we have seen, in fact, in presence of congestion four TCP connections use the bandwidth better than a single TCP connection (as suggested in the literature [295]), which allows for better bandwidth measurements. But, saying that four connections use the bandwidth more efficiently than a single connection is the same as saying that four connections are less sensitive to losses, and to latency, than a single connection. Therefore, a single connection is a better tool (compared to four connections) to spot low quality networks.

Slowly these concepts become clearer and clearer, but in May 2011 were not clear enough. The paper that shaped our awareness, in fact, was the one by Bauer, et al., [180], studied in January 2012. Therefore, even if our subsequent choices (i.e., returning to two connections) appear *ex post* led by the desire to measure more accurately the quality than the broadband speed, they were instead motivated, as we shall see, by more practical reasons. Yet, we already had enough insight to understand that more servers were needed to perform better measurements. For this reason, at the end of May, we applied to M-Lab [206], which had 60+ servers around the world at the time.

Neubot 0.3.7 Released on May 20, 2011, Neubot 0.3.7 included a Speedtest that used only two connections, i.e., $\nu = 2$. In this new version of the Speedtest we also increased the test duration and we made the test less aggressive. All these changes were made to solve a specific, practical problem: using four connections it was hard to control the duration of the upload test phase, especially in home networks, where the upload phase lasted for up to ten times the expected duration (one second).

With the insight of 2014, it is obvious what was going on: the upload time was huge because of the bufferbloat (the large queuing delay typical of home networks, better described in Section 7.2.4): it took time, in fact, to empty the large queue created at the home router. At the time, however, we had no clue of the bufferbloat, and we found cheaper to return to two connections and to increase fivefold the expected duration of the test. Retrospectively, the former change contributed to reduce the *bufferbloat* created on the router, and the latter change guaranteed few low-speed peaks. In fact, as we shall see in Section 6.6, being the transfer five times longer, the impact of losses during the slow-start transient on the

average speed was five times smaller (so, the result was more predictable, because losses have higher impact during slow start than when TCP is at the equilibrium).

More specifically, *was_too_short* became $\delta < 3$ (the infinite-loop check, $\eta < 4$, was removed in Neubot 0.3.7 because, meanwhile, the server was updated to serve arbitrary number of bytes, therefore infinite loops were not possible anymore[4]). As regards *adapt_nbytes*, it was amended as follows: when δ is less than one second, *nbytes* is doubled (as before); otherwise, *nbytes* is linearly scaled such that the elapsed time of the next test should be five seconds (assuming stable network conditions). We chose linear scaling because we experimentally noticed that the exponential scaling was too aggressive when δ was greater than one second. As the reader has probably noted, with the new algorithms we scale *nbytes* to obtain $\delta = 5$ (because of *adapt_nbytes*), but we avoid repeating tests in which $\delta \geq 3$ (because of *was_too_short*). We do that because we do not want to repeat a test that lasted three seconds or more: its repetition, in fact, would take at least three more seconds, leading to a test that would be too long (given that Neubot is a tool running in the background and given that Speedtest downloads data at maximum speed).

In addition Neubot 0.3.7 also modified σ (the size of the socket receive buffer) to be 256 KB (previously it was zero, meaning that the operating system was free to set, and possibly scale, the size of such buffer). We did this change because we observed that different operating systems scale the receive buffer differently during the slow start[5] and we wanted to eliminate such possible confounding factor. In particular, we used 256 KB because we noticed that the OpenBSD receiver buffer could not grow above 256 KB (unless one uses root privileges to invoke sysctl and raise such limit).

6.5 Performance of Selected ISPs in the Turin Area

In this Section we show how we used Neubot to benchmark the performance of the most-widely-used ISPs in the Turin area. To this end, we use the data collected in the period from May 30, 2011 to September 13, 2011 (the raw data is, of course, available online on the data.neubot.org website [103]). The majority of the Neubot instances shown in this analysis had version numbers between 0.3.7 and 0.4.1. Therefore, the ISPs were benchmarked using mainly the two-connection fixed-receiver-buffer methodology. Even if two connections are used, it is clear per the previous discussion on broadband speed measurements, that the results are only correlated with the broadband speed and that they are also correlated with the ISP quality (i.e., the base latency, its variations, and the average loss rate). Let us also remind that the Speedtest methodology does not account for confounding factors such as poor wireless connectivity and the activity of other users that share the same home network (more on possible confounding factors in Section 6.6).

[4]Moreover, we also added a 'watchdog timeout' mechanism that guarantees that a test cannot run for more than a given number of seconds.

[5]Slow start is the state in which TCP estimates the maximum number of bytes that the network can hold by exponentially increasing its window. For more on TCP, please refer to Chapter 7.

	Whole dataset	Turin area
Day of the first test	30-05-2011	30-05-2011
Day of the last test	13-09-2011	13-09-2011
Number of tests	947,729	80,196
Number of Neubot instances	1,054	166

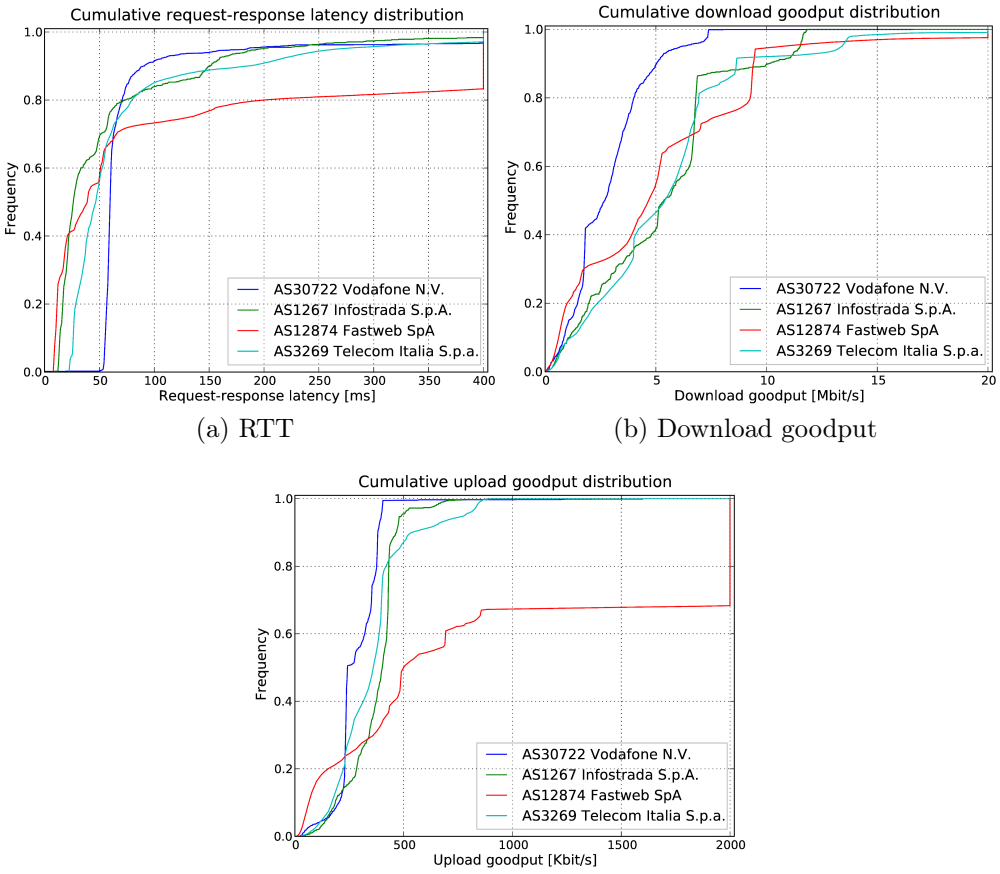
Table 6.2: Main data for the Turin-area measurement campaign.

Autonomous System	Measurements	Neubots
AS2594 CSI Piemonte	77	1
AS44957 OPITEL AS number	45	2
AS8968 BT Italia S.p.A.	1234	2
AS35612 NGI Spa	11	3
AS35719 TEX97 S.p.a	573	3
AS24608 H3G S.p.A.	18	4
AS16232 TIM (Telecom Italia Mobile) Autonomous System	90	4
AS8612 Tiscali Italia S.P.A.	555	7
AS137 GARR Italian academic and research network	26200	25
AS30722 Vodafone N.V.	1162	8
AS1267 Infostrada S.p.A.	794	24
AS12874 Fastweb SpA	24735	44
AS3269 Telecom Italia S.p.a.	24702	65

Table 6.3: Turin-area Neubot data per Autonomous System.

Table 6.2 shows the number of tests, the number of Neubot instances and the date of the first and last test for the whole dataset and for the subset of instances geolocated in the Turin area. To geolocate the Neubot instances and find those in the Turin area, we used MaxMind GeoLite [11], both for geolocation and for the identification of the Autonomous Systems (routing entities that belong to a given ISP). We focus on the Turin area because our test server is located in the Torino-Piemonte Internet eXchange (TOP-IX), therefore, we show only measurements performed by 'nearby' (in terms of RTT) instances. Of course, having enough servers, the analysis performed on the Turin area can be repeated for other regions and countries to build a per-geographic-location per-provider results map. Table 6.3 shows the number of measurements and the number of Neubots geolocated in the Turin area for each Autonomous System (AS). In this Section we show the results of the four commercial providers with more tests and more users in the Turin area: Vodafone, Infostrada, Fastweb and Telecom Italia. GARR, the Italian universities network, is not included, even if it has many tests and users, because its performance are not comparable to the ones of commercial providers, having much more access link capacity (e.g., in our campus we have access speeds ranging from 100 Mb/s to 1 Gb/s). Of course, since Neubot data is publicly available, one can build its own qualitative analysis.

Figure 6.3a shows the cumulative distribution of the request-response latency for the tests performed in the Turin area. The distribution is plotted in the range 0-400 ms and



(a) RTT (b) Download goodput

(c) Upload goodput

Figure 6.3: Empirical cumulative distribution of request-response latency, download and upload goodput for the tests performed in the Turin area in 2011.

greater values are assigned to the 400-ms bin. We use request-response latency because the connect-time estimation was buggy in old versions of Neubot, as mentioned in Section 6.2. About 20% of Fastweb tests have latency higher than 400 ms: this is caused by 20 Neubots that consistently report very high latency. This anomaly seems not to be caused by the fiber vs. ADSL split of Fastweb customers: there are many Neubots with less than 1 Mb/s upload both in the more-than-400-ms camp and in the other camp. We can speculate (but we cannot confirm) that those Neubot instances were installed in heavily used access networks with high queuing delay.

Figure 6.3b shows the cumulative distribution of the download goodput for the tests performed in the Turin area. The distribution is plotted in the range 0-20 Mb/s and greater values are assigned to the 20-Mb/s bin. The goodput is calculated at the receiver, dividing the number of bytes transmitted by the time elapsed from sending the request to receiving the whole response. If we exclude Vodafone, which has a single 7-Mb/s-download–512-kb/s-upload commercial offer, results are collected by users with different broadband speeds and

contracts; however, we can identify peaks that are linked to typical access-networks speeds, e.g., there is a 7 Mb/s peak in the Telecom Italia distribution. In this regard, it would be interesting to classify the results depending on the highest measured speed and plot the contribution of each class separately. Another interesting fact is that very few users appear to download at speeds closer to 20 Mb/s. Since in Italy there are commercial offers that provide such access speed, it would be interesting to understand whether such speeds are not reached because (a) very few Neubot users have 20 Mb/s access links; (b) the speed is advertised but not actually delivered; (c) the methodology of our Speedtest does not identify 20 Mb/s access networks (e.g., because the latency is high and the test is too short for TCP to reach the equilibrium and use all the available bandwidth).

Figure 6.3c shows the cumulative distribution of the upload goodput for the tests performed in the Turin area. The distribution is plotted in the range 0-2 Mb/s and greater values are assigned to the 2-Mb/s bin. The goodput is calculated at the sender, dividing the number of bytes transmitted by the time elapsed from sending the request to receiving the whole response. Excluding Vodafone, results are collected by users with different broadband speeds. For example, 35% of Fastweb tests are above 2 Mb/s, clearly due to Fiber-To-The-Premises (FTTP) connections (at the time, in fact, Fastweb had two different access networks, one using the ADSL and the other using FTTP).

To wrap up: we show that Neubot can be used to analyze the performance of ISPs in a selected area (in our case, the area close to our measurement server). As anticipated, the performance is correlated both to the broadband speed and to the ISP "quality". The measured speed, in particular, seems correlated with the advertised broadband speed in many cases (e.g., the Telecom-Italia 7-Mb/s download speed). However, there are also more complex cases, such as the 20 Mb/s case: we do not know whether 20 Mb/s download speeds are not common because 20 Mb/s are advertised but not delivered, or because our methodology is not adequate to measure 20 Mb/s ADSL connections (for example, because the test is too short). In the next Section we will discuss the Speedtest methodology in general and, in particular, we will also provide a tentative explanation of the 20 Mb/s dilemma: since we force the receiver's buffer to 256 KB, which is too small, we cannot correctly measure connections with bandwidth as high as 20 Mb/s.

6.6 Discussion of the Speedtest Methodology

In this Section we discuss the pros and the cons of the Speedtest methodology, focusing in particular to the post-0.3.7 Speedtest. In doing so, we will incorporate the comments that we received during the M-Lab review of Neubot. In such review, in particular, the researcher that was in charge of commenting our scientific methodology was Matt Mathis, who is well known in the network-performance community for, e.g., his work on TCP performance [250], Forward Acknowledgments [248], and the Web100 project [251].

RTT Estimation Let us start by discussing the accuracy of the procedure we use to estimate the RTT. The 'time required to connect' approximation of the RTT is acceptable

if neither the `SYN` nor the `SYN|ACK` segments are lost [253], and if the two operating systems are not heavily loaded, so that the network delay is at least one order of magnitude larger than the segment processing delay. Compared to the time required to connect, the 'request-response latency' is less accurate, because there is always some application delay involved and because there is the overhead caused by the HTTP messages, yet it allows one to collect more than a single sample per connection. Of course, both estimates are performed at the beginning of the connection, therefore they may not reflect the actual RTT during the data transfer (more on this topic in Section 7.2.4, where we define the *bufferbloat*).

Another RTT-related problem is the one of comparing the performance. On the one hand, the Mathis model for TCP performance says that the goodput is inversely proportional to the RTT [250]. Therefore, the comparison is possible for similar RTT ranges but more problematic when the RTT significantly differs. Moreover, the behavior of TCP is typically modeled in terms of 'rounds', where the round duration is, indeed, the RTT [261]. Therefore, connections with different round-trip times run the test for the same number of seconds but certainly not for the same number of rounds. That is, the connections that run for more rounds perform a more thorough network test than do connections running for less rounds. In other words, if the probability of losses is below a certain threshold, connections that run for a short number of rounds may not experience any loss.

The previous observation on the number of 'rounds' also allows us to explain why low-speed spikes were more frequent in one-second tests than in five-second tests. Tests running for just one second, in fact, spent a larger fraction of their time in the slow-start state than tests running for five seconds. In turn, the slow start is a state in which the bandwidth is used less efficiently than in the equilibrium state, called congestion avoidance; moreover, in slow start TCP is more sensible to losses. We can conclude that, with one second tests, there was more variation because (a) the occasional losses that occurred during the slow start had a higher impact on the goodput and (b) one-second-tests spent a larger fraction of their time in the slow start (compared to five second tests).

Goodput Estimation The upload goodput estimate is measured at the sender, for implementation simplicity. However, it would be more robust to measure the upload speed at the receiver, for a number of reasons. First, measuring at the receiver smooths possible peaks, such as the ones caused by drop tail queue management at the bottleneck (which often is the home router). Moreover, more generally, the measurement at the receiver conveys more information, because the burst of packets sent by sender is processed by all the intermediate ISPs that can modify its characteristics (e.g., the inter-packet gap). In addition, measuring at the sender has the problem that the time needed to empty the socket buffer – a non negligible time, in particular, when the queuing delay is high and the bandwidth is low (see *bufferbloat* in Section 7.2.4) – is not measured.

Another goodput-related methodological issue is that we always underestimate the actual network goodput. Our methodology, in fact, always yields an underestimate of the goodput because the measurement starts when the requester sends the request; therefore the elapsed time includes at least one idle round-trip time, the one in which the requester

is waiting for the response to arrive. The impact of such extra, idle RTT was higher with one-second measurements than it is with five-seconds ones. Assuming a round-trip time of 100 ms, in fact, the idle time was 10% (or more) with one-second measurements, while it is 2% (or more) with five-seconds measurements (assuming that the measurement lasts for exactly five seconds, which, of course, is a simplifying assumption).

Multiple Connections As regards the multiple-connections model, we already anticipated above that it has the following advantage: the measured goodput is closer to the throughput. The available bandwidth, in fact, is used more efficiently, because each connection has a smaller window, therefore, it takes less round-trip times to recover after a loss; moreover, multiple connections are more efficient because a connection may grow its congestion window and take more bandwidth when another experiences a loss. As a consequence, measuring a congested network with more concurrent connections yields a better goodput. This is not desirable if one's goal is to measure the quality, however, because the measurement should seek to penalize (not reward) congested access links.

Fixed-Size Receiver Buffer Similarly, we already discussed the advantages of using a fixed-size receive buffer, i.e., that all operating systems should have the same behavior, because there are no differences related to the algorithm that the system uses to automatically scale such buffer (provided that it is scaled). Moreover, knowing in advance the buffer size simplifies the analysis of the results, because it is easier to classify buffer-limited TCP connections. On the other hand, however, the problem with this strategy is that most high-speed high-delay connections are likely buffer limited, especially in countries like Korea and Japan where it is common to have 100 Mb/s access links. This fact is not desirable because, again, uncongested fast access links should be rewarded, not penalized. In turn, this fact can be a reasonable explanation of why there were very few 20 Mb/s download speed results in Section 6.5.

Other Confounding Factors Building on the excellent work by Bauer, et al., [180], let us remark that our methodology does not account for many confounding factors that may reduce the goodput. It assumes, in fact: that the user is not heavily using her computer and/or her access link; that no other users are heavily using/sharing the same access link; that the WiFi network (if any) is not lossy; that there is no congestion in the backbone; that the test server is not overloaded.

Concluding Remarks The two most immediate conclusions originating from this analysis were that, if one's objective is to clearly distinguish between good and bad access networks, a single connection should be used and the operating system shall be free to automatically scale the receiver's buffer. Precisely those two changes were implemented in subsequent releases of Neubot, as described in the following Section.

Moreover, this analysis was also the starting point of our subsequent research activities. In fact, as we will see in the next Chapter, since the M-Lab review, we spent time to

Speedtest parameters and functions	Neubot version			
	[0.3.0–0.3.4]	[0.3.5–0.3.6]	[0.3.7–0.4.1]	[0.4.2,)
ν	2	4	2	1
was_too_short	$\delta < 1$ and $\eta < 4$		$\delta < 3$	
$adapt\_nbytes$	if $\delta < 1$ then $nbytes * 2$ else $nbytes$		if $\delta < 1$ then $nbytes * 2$ else $nbytes * 5/\delta$	
σ	0		256 KB	0

Table 6.4: Evolution of Speedtest methodology (see Section 6.2 and Algorithm 1 for a description of the Speedtest functions and parameters).

research the links between the application-level results of Neubot and the variation of key network-level performance parameters (e.g., the packet loss rate).

6.7 Changes after the M-Lab Review

We concluded the previous Section saying that, after the M-Lab review, it was clear that, to estimate the quality of access networks, one should use a single connection and let the operating system handle the socket buffers. Doing that, in fact, one is more confident that high losses lead to a low goodput and that high bandwidth leads to a high goodput. Therefore, with Neubot 0.4.2 we significantly changed the Speedtest again: Neubot 0.4.2 shipped with a single-connection Speedtest in which the fixed-size receiver buffer was not enforced anymore. The single-connection Speedtest reduced the range of bandwidth-delay products that we could serve (in fact, during the 0.3.7 development process, we decided to keep two connections exactly for this reason); however, during the development of 0.4.2, we already knew that Neubot was about to be added to M-Lab: given that M-Lab had, at the time, over 64 servers around the world, the range of bandwidth-delay product was not going to be a concern anymore.

We conclude this Section with Table 6.4 that summarizes all the changes that occurred to the Speedtest from the inception to Neubot 0.4.2 (which was the last version of Neubot, as of February 2014, in which the Speedtest methodology was changed).

6.8 Concluding Remarks

In this Chapter we described the history of the HTTP Speedtest (or, simply, Speedtest) Neubot test. This test was the first test that we wrote for Neubot and was inspired by the namesake test available at Speedtest.net and designed by Ookla [136]. The initial objective of the test was to characterize the Internet connection of a user by estimating the round-trip time (RTT), the downstream link speed and the upstream link speed. We added this test to Neubot because – to validate that its architecture fitted the purpose of running network experiments from the edges – we needed to start with measuring a performance metric that was already widely measured, so that we could have a yardstick for our results.

To this end, we designed and developed a lightweight test that estimates: the RTT using (a) the time that the connect() system call takes to complete and (b) the request-response

application-level delay of small requests and responses; the download and the upload speed of the access network by performing, respectively, a HTTP download and a HTTP upload. To ensure that the download and the upload speeds approximated the bandwidth, we only considered the results of tests originating from Neubot instances geographically close to our measurement server; we repeated the download and the upload, requesting or sending more bytes, until they run for at least one second. The one-second-long target for the HTTP transfers was chosen because Neubot is a background tool that shall use few network resources (we were, of course, ready to run longer tests, if needed).

In early 2011, a first experiment campaign confirmed that the Neubot architecture fitted the purpose of running collaborative measurements from the edges. We selected the subset of the available results that was reasonably close to the only server that we had at the time (master.neubot.org provided by TOP-IX and located in Turin) and we verified that the average download speed was comparable to the speeds measured by Ookla Speedtest.net and YouTube for, respectively, Italian home-access networks and users based in Turin.

After concluding that the Neubot architecture was adequate, we turned our attention to the initial Speedtest implementation. We tried to understand why the measured speeds presented a certain degree of variation and, also prompted by comments from a Neubot user, we ran many experiments. In conclusion, we stabilized the measured speed raising the expected Speedtest duration to five seconds and fixing the socket reiceiver's buffer to 256 KB, to avoid operating-system-dependent differences which influenced the results.

After such methodology changes, we run a second experiments campaign during the summer of 2011. We evidenced that Neubot could be used to collaboratively estimate the performance of the most-widely-used ISPs in the Turin area, with results that show peaks corresponding to broadband speeds made available by such ISPs.

We also noticed that there were few results for which the download speed was around 20 Mb/s, at the time the top-tier ADSL available in Italy. To understand whether the lack of 20 Mb/s results was a problem of our methodology, we invested more time to study its pros and cons. Also, at the time, Neubot was being reviewed by M-Lab, a review that produced interesting comments on our methodology. In short, our studies and the comments received during the M-Lab review suggested to change the methodology again: we removed the fixed-size receiver buffer and we used a single connection, two changes that made the Speedtest more sensitive to quality (i.e., after such changes a high measured goodput was more correlated with a good access network than before).

Moreover, our studies and the M-Lab review had an impact not only on the Speedtest methodology but also on the direction in which the Neubot project and our research evolved. In autumn 2011, in fact, we started a one-year-long effort to understand more thoroughly the link between application-level measurements and key network-level parameters (e.g., the packet loss rate), as we will see in the following Chapter.

As a final remark (and for future memory) let us quote here a sentence from the conclusions of our AICA 2011 paper [175], one of the two papers in which we described the Speedtest test: "future versions of Neubot should dedicate one test to estimate the quality of the network, using just one connection, and another test to approximate the real broadband connection speed, using as many connections as needed." We never did that: in

fact, we abandoned the multiple connection Speedtest because our research path evolved in another direction. However, it would be interesting in the future to resume the multiple connection Speedtest, to increase the array of measurements performed by Neubot.

Chapter 7

Experiment #2: Application-level TCP Packet-Loss-Rate Estimation

This Chapter describes two models (Inverse Mathis and Likely Rexmit) for estimating the packet loss rate (PLR) experienced by a TCP connection using application-level measurements. The Inverse-Mathis and the Likely-Rexmit models are important for Neubot for two reasons. Firstly, because they are designed to help network-performance-tools such as Neubot to estimate the PLR, a key performance metric, using simple application-level measurements. Secondly, because they were validated with data collected by (among other things) Neubot-based large-scale network-performance experiments. In other words, the models are useful to improve the performance of Neubot and the study of the Inverse-Mathis and the Likely-Rexmit models was only possible because Neubot provided us with a flexible platform to run network tests from the edges.

This Chapter is organized as follows: in Section 7.1 we say what makes application-level PLR estimation an interesting research topic; in Section 7.2, we briefly describe the basic TCP concepts used in the Inverse-Mathis and Likely-Rexmit evaluation; in Sections 7.3 and 7.4, we describe and we evaluate the Inverse-Mathis model, using as reference the paper presented at the 2nd Workshop on Measurements Up and Down the Stack [177] as well as subsequent experiments and studies; in Sections 7.5 and 7.6 follows up the Likely-Rexmit-model description and comparison with Inverse Mathis, which follows very closely the journal paper published on the Computer Communication Review [178]; finally, in Section 7.7 we make our concluding remarks.

7.1 Motivation

In Chapter 1 we used the end-to-end principle to link the notion of network neutrality to the original, intended design of the Internet. In practice, the end-to-end principle was implemented by dividing the network in layers: the link layer (Ethernet, ATM, etc., common to end-hosts and routers), the network layer (IP, common to end-hosts and routers), the transport layer (TCP and UDP, used by end-hosts only), and the application layer (SMTP,

FTP, HTTP, etc., used by end-hosts only). In this Chapter, we will see that the layered design of the Internet (and, in particular, the transport layer) has now become an obstacle for those who wish to increase the transparency of the network.

In the early days, when Internet-service providers (ISPs) barely cared about the transport and application layers, it was wise to hide the complexity of TCP behind the opaque socket interface. This way, every program only needed to link to the C library and to issue the proper system calls to be able to use the network. Nowadays, on the contrary, we are in the paradoxical situation that ISPs potentially know much more on the state of TCP flows than the applications using such flows. And note that we are in this situation precisely because, on the one hand, the sockets are opaque and because, on the other hand, it is now possible to infer the state of the transport and application protocols from passive observation of the network traffic (e.g., there is a work by Mellia, et al., [254] that describes a technique to reconstruct the state of TCP to classify anomalies). Moreover, as seen in Section 1.2, the network is not only a passive observer of the layer four state, but also a reactive agent that can change its behavior in accordance with what was transmitted. For example, the network can flag certain flows as undesirable and route them over very congested paths, using the Multiprotocol Label Switching (MPLS) [158] technology.

Therefore, we say that that the layered model – which helped to implement the separation of concerns that is the cornerstone of the end-to-end principle – is not serving us Internet folks well anymore, because it now acts like a barrier that prevents applications from accessing information useful to characterize the active behavior of the network.

As a proof that the status quo is unsatisfactory, we can cite a few efforts that either strive to make TCP more transparent or incorporate the entire functionality of TCP into the application. As regards efforts to make TCP more transparent, we are referring in particular to the modified Linux kernels provided by the Web100/Web10g project, whose kernels allows one to gauge, e.g., the current congestion window and to retrieve the number of congestion signals received [251]. As regards incorporating the TCP functionality into the application, we would like to cite the Low-Extra Delay Background Transport (LEDBAT), an application-level TCP replacement that not only reacts to the well-known congestion signals (triple duplicate ACKs, timeouts, etc.) but also considers an early-congestion signal the excessive increase of the forward-path queuing delay. Notwithstanding their radically different approaches, both Web100/Web10g and LEDBAT have in common that they allow the developer to "route around" the opaque sockets to get much more information on the network layer events (e.g., increases of latency, bursts of losses).

In fact, both Web100/Web10g and LEDBAT were successfully used to solve problems that were not easily solvable using the standard application-layer tools only: Tirumala, Cottrel and Dunigan proposed to modify Iperf (a well-known network-performance measurement tool [154] that estimates the end-to-end bandwidth using a *20-seconds long* TCP or UDP network test) to use Web100/Web10g data for estimating the available bandwidth just after TCP exited the slow start (the TCP state in which the available bandwidth is estimated), thereby reducing the estimation time of up to 94% and the network traffic of up to 92% [296]; LEDBAT was specifically invented by the BitTorrent developers to reduce the amount of queue that BitTorrent clients used to create on the bottleneck, thereby

improving the average quality of experience of heavy file-sharers [194] (one could object that the same benefit could have been achieved with a TCP implementation of the core LEDBAT algorithms, however the application-layer implementation of LEDBAT arguably entailed less coordination costs than convincing all the major operating systems to merge the LEDBAT algorithms into their kernels).

In addition to helping to write better applications, the availability of network-level data is also crucial for studying network neutrality, which is why it is relevant to us. As discussed in Chapter 1, in fact, the only winning move to rebalance the information asymmetry that characterizes the network-neutrality debate is to provide users with information to which they do not currently have access, e.g., one thing is to know that one's download speed was 23 Mb/s, another thing is to know which network-level performance metric determined such speed. Of course, the average user may not necessarily be able to fully understand this kind of information, yet consumer associations, policy makers and researchers studying the problem will certainly be pleased to have such information at hand.

Strongly motivated by the fact that network-level data is crucial to study network neutrality, in this Chapter we describe how we contributed to this topic. Compared to the examples presented above, we attacked the problem from a different angle. We studied, in fact, what information on the packet loss rate experienced by a TCP flow (a key network-level performance metric) one can extract from application-level measurements. We were interested in application-level measurements for two reasons. Firstly and foremost, because application-level measurements are more scalable (in terms of users that accept to run such measurements) than measurements that require a modified kernel. Secondly, because we were interested to study measurements techniques that are applicable to Neubot, which as we know works at the application level. Moreover, let us add that application-level measurements are not necessarily alternative to the more-precise kernel-level measurements, rather application-level measurements can be one of many signals that could feed a layered measurement system (such as the one envisioned by Palfrey and Zittrain [262]) capable of collecting insights from multiple sources and able to use coarse-grained data to trigger more-specific measurement tools when needed or useful.

We will return to this topic of a multi-layered measurement system in Chapter 10; now it is time to guide the reader through the fundamental TCP concepts used in the evaluation of the Inverse-Mathis and the Likely-Rexmit models.

7.2 A Brief Overview of TCP

In this Section we provide a brief overview of TCP (the transmission control protocol) that encompasses the most important concepts needed to understand the evaluation of the Inverse-Mathis and the Likely-Rexmit model. Our description of TCP is not meant (of course) to be exhaustive, rather we aim to gather the main TCP concepts in the same place, along with pointers to the literature.

The rest of this Section is organized as follows: at first, we describe the TCP basics; next, we provide a brief history of TCP; then, we describe one alternative TCP flavor

that we will meet later in this Chapter, TCP Cubic; finally, we provide the definition of 'bufferbloat', which is widely used from Section 7.3 onwards, along with a few other useful definitions.

7.2.1 The Basics: Packets, ACKs and SACKs

TCP is a reliable protocol that is based on the concept of sending packets (or segments) and receiving back acknowledgments (in short, ACKs). Each packet is identified by a sequence number and each ACK carries a confirmation sequence number. The ACKs are cumulative, therefore, if the sender sends, e.g., packets 1–10 and only packet seven is lost, a receiver may generate a single ACK confirming packet five (and hence also packets 1–4) plus four ACKs that confirm packet six (the three duplicate ACKs indicate that three packets with sequence number greater than 7 were received).

A more advanced ACKing scheme, called selective ACKs (SACKs), was introduced at the beginning of the 1990s [249] and is available as an option. When the SACK option is enabled, the receiver provides the sender with a more clear picture of which packets were received and which packets were not received. To continue the previous example, with the SACK option enabled, the receiver ACKs packet 9 with an ACK for 6 (as before) that, moreover, includes a vector (called 'SACK vector') containing 8 and 9 (i.e., the sequence numbers of the greater-than-six packets received so far).

Because they provide the sender with more information, SACKs help the sender to respond to congestion (i.e., packet losses) in a more efficient way. Yet, SACKs were added to TCP relatively late, after a simpler theory for dealing with congestion had already been researched, validated and implemented into BSD Unix (which had been the reference TCP/IP implementation for a long time). The history of how TCP evolved from the 1981 seminal RFC793 by Jon Postel [264] to TCP "NewReno" (one of the most widely used TCP flavors) with SACKs is the covered by the next Section.

7.2.2 History of TCP from RFC793 to NewReno with SACKs

According to RFC793 [264] the throughput of TCP is limited by the buffer space available at the receiver. Subsequent revisions of the standard added improved mechanisms to control the injection of traffic into the network, however the basic mechanism of operation is still used nowadays. Packets and ACKs carry the so-called receiver-advertised window (RWND) that communicates to the sender the buffer space available at the receiver. To control the flow, TCP does not allow the sender to send more than RWND packets before receiving a cumulative ACK (and, hence, an updated RWND snapshot) from the receiver. The sender also estimates the 'typical' time elapsed from sending a packet to receiving the corresponding ACK, called round trip time (RTT). The RTT estimate, in turn, is used to compute the retransmit timeout (RTO): if no ACK for packet P is received within an RTO, the sender declares lost all the packets transmitted after packet $P-1$. After a timeout, the historical TCP transmitted a new, RWND-sized batch of packets starting from P; modern TCPs, instead, are more conservative as we shall see in the following.

The reason to be more conservative after a timeout (as well as when you transmit for the first time) is that the buffer space available in the network may well be lower than the RWND: in such case each transmission of RWND packets results in potentially large burst of losses. This phenomenon was, in particular, very problematic for long-distance connections between universities, which at the time were implemented using slow data links. To avoid sending more packets than the network can handle, in 1988 Jacobson improved TCP to make it congestion aware [228]. The general idea of the improved TCP (today commonly called 'TCP Reno' because it was released with 4.3BSD 'Reno') is that the sender should obey to a packet conservation principle: once the connection is established, the sender should aggressively estimate the maximum number N of packets that the network can handle; once N is known, the job of the sender is to keep N packets 'outstanding' (i.e., sent but not acknowledged yet). To this end, the sender shall send a new packet only when an old packet is ACKed. Moreover, because the network conditions can change, the sender shall also (a) slowly inflate N to probe for more bandwidth and (b) quickly deflate N to respond to 'early congestion signals' (i.e., signals from the network suggesting that either the sender slows down or there will be a timeout caused by a burst of losses).

To implement the strategy described above, Jacobson introduces (among other things) a new variable, the congestion window (CWND), that keeps track of the maximum number of packets that the network can handle, N. Like the RWND, the CWND controls the maximum number of packets that TCP can send before it receives an ACK. However, while the RWND is controlled by the receiver, the CWND depends on the network. In particular, Reno exponentially inflates the CWND to aggressively estimate N, linearly inflates the CWND to probe for more bandwidth, and halves the CWND to respond to early congestion signals (leading to an exponential decay if congestion persists). As regards the definition of 'early congestion signal', Reno halves the CWND when it receives three duplicate ACKs for packet P, meaning that $P + 1$ was probably lost but the network is not down, for three packets with sequence number larger than $P + 1$ were received.

A simplified Reno finite state machine (inspired to the description of the Linux TCP implementation by Sarolahti and Kuznetsov [279]) is described in Figure 7.1. At the beginning TCP is in the OPEN state, the CWND value is two (packets), and the slow start threshold (SSTHRESH), used to decide when to switch from the initial aggressive behavior to the steady-state behavior, is set to half the send buffer space (in packets). In the OPEN state TCP opens the CWND: at the beginning the CWND is increased exponentially (+1 for each ACKed packet); instead, when the the CWND grows larger than the SSTHRESH, it is opened linearly ($+1/CWND$ for each ACKed packet). As said, when Reno receives three duplicate ACKs for packet P, it assumes that packet $P + 1$ was lost. To respond to such a loss, Reno sets CWND and SSTRHESH to half the current CWND, it retransmits the lost packet, and it enters into the RECOVERY state. In such state, TCP seeks to keep a constant number of outstanding packets, by sending (if possible) a new packet for each subsequent duplicate ACK, until packet $P + 1$ is ACKed. At that point, Reno returns to the OPEN state. At any time, if there is a timeout for packet Q, TCP sets the SSTHRESH to half the CWND and the CWND to one packet, it retransmits packet Q, and it enters into the LOSS state, in which it stays until Q is ACKed.

Of course, the finite state machine described above goes hand-in-hand with the RWND-based flow-control mechanism introduced earlier. In other words, at any given time the number of outstanding packets shall be lower than the minimum of the RWND and the CWND. That is, depending on the circumstances TCP performance are either controlled by the CWND or by the RWND. As we will see later, this fact is related with the concept of 'application-limited TCP' (introduced in Section 7.2.4) that, in turn, is quite relevant for our discussion of the accuracy of the Inverse-Mathis model.

Returning to the finite state machine, the description we provided was simpler than the one provided by the standard, RFC5681 "TCP congestion control" [171]. Moreover, RFC5681 describes TCP in terms of algorithms, while we used a finite-state-machine point of view (in our opinion, simpler). In particular, the following are the most notable differences between our description and the RFC: what we described as the OPEN state is described by RFC5681 as two diverse algorithms called 'slow start' (when the CWND is increased exponentially) and 'congestion avoidance' (when the CWND is increased linearly); the transition from the OPEN to the RECOVERY state (in which the CWND is halved) is called 'fast retransmit algorithm', while the RECOVERY state is described as the 'fast recovery algorithm'; our description of how the CWND is halved to respond to early congestion signals is simplified with respect to the one provided by the RFC, because we neglected some implementation details; our description of the LOSS state does not describe the case of multiple subsequent timeouts; our description of how the CWND is managed is only valid for full-size packets (the maximum packets size is agreed by the sender and the receiver when the connection is established).

On top of the above algorithms, many modern Reno implementations add one extra modification, commonly called 'NewReno', initially proposed by Hoe in 1995 in his Master's thesis [225], and now documented by RFC6582 [224]. This modification changes the way in which the recovery works, by teaching TCP how to better respond to 'partial acknowledgments', i.e., "ACKs [, received when the TCP state is RECOVERY,] that cover new data, but not all the data outstanding when loss was detected" [224].

Moreover, TCP NewReno is typically coupled with the SACKs mechanism introduced above: when the SACK option is enabled, NewReno halves the CWND not only when a triple-duplicate ACK is received, but also when the SACKs vector indicates that packet P was not received, but three or more packets with sequence numbers $S_i > P$ were received. In the interest of brevity, in the following we call 'early congestion signal' not only the congestion signal originated by triple-duplicate ACKs but also the congestion signal inferred by counting the number of received packets listed in the SACKs vector.

7.2.3 Cubic TCP

As said, TCP NewReno linearly inflates the CWND for each ACKed packet and halves the CWND to respond to an early congestion signal. However, TCP NewReno is far from being the only available TCP flavor; many researchers, in fact, proposed alternative TCP flavors that use diverse congestion avoidance strategies to improve the performance in, e.g., high-speed networks [214, 234, 291, 222] and wireless networks [190, 246].

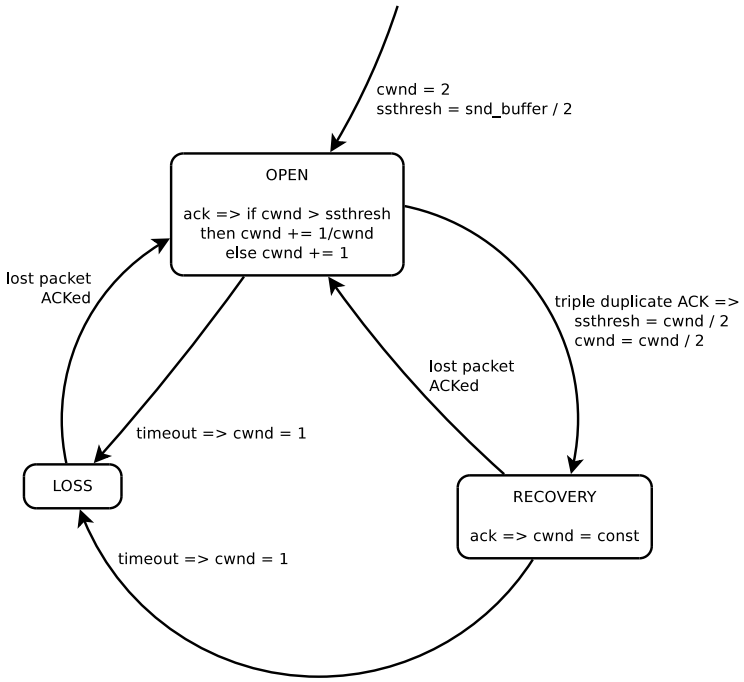


Figure 7.1: A simplified TCP Reno model.

As far as our discussion of the Inverse-Mathis and Likely-Rexmit models is concerned, the most relevant alternative flavor of TCP is Cubic [222]. We are particularly interested in Cubic because it is the default TCP congestion-control algorithm on Linux since version 2.6.18. (We are less interested in other TCP flavors because the other major operating systems, Windows, MacOS and BSD, use NewReno by default.)

In the following, we describe how Cubic works and we list the major differences between Cubic and NewReno (the typical evolution of the Cubic CWND, by the way, is well described by connection #60 of Figure 7.4a):

1. after an early congestion signal, the CWND is multiplied by 0.8;

2. then, the CWND growth follows a cubic function starting from $0.8 \cdot old\_cwnd$ (i.e., the value set after the last loss) and having inflection point at old_cwnd, i.e., the CWND value that should trigger the next loss (assuming constant network conditions);

3. at the beginning, the cubic function is concave and will quickly grow, then it will flatten near the inflection point (thus, Cubic uses the bandwidth more efficiently, because it spends more time than NewReno near the inflection point);

4. moreover (if no loss occur because the network can now handle more packets) the cubic function will overcome the inflection point, will become convex, and will start to grow very fast, allowing Cubic to aggressively probe for more bandwidth;

5. but, to guarantee that Cubic does not steal bandwidth from the standard TCP, Cubic implements a TCP-friendliness mechanism that guarantees that the Cubic CWND does not grow faster than the NewReno CWND would do in similar conditions.

As regards 5., Cubic is specifically designed to be TCP-friendly in networks with a short RTT as well as in networks with a small bandwidth-delay product (BDP). Simulations carried out by the Cubic authors, in fact, show that the average Cubic CWND equals the average NewReno CWND for a wide range of loss rates (from 0.01 to 10^{-6}) when the RTT equals 10 ms; but, when the RTT equals 100 ms and the loss rate is smaller than 0.001, the average Cubic CWND is much larger than the NewReno one [222].

We will return to Cubic in a while, because many of the experiments that we run involved Linux boxes running TCP Cubic. However, before we proceed, let us provide some more TCP-related definitions, including the one of 'bufferbloat', a concept that is used to discuss the accuracy of the Inverse-Mathis model.

7.2.4 The Bufferbloat (and Other Useful Definitions)

In this Section we provide a number of definitions that are widely used throughout this Chapter, including the definition of 'bufferbloat' that is used, in particular, when we will discuss the accuracy of the Inverse-Mathis model.

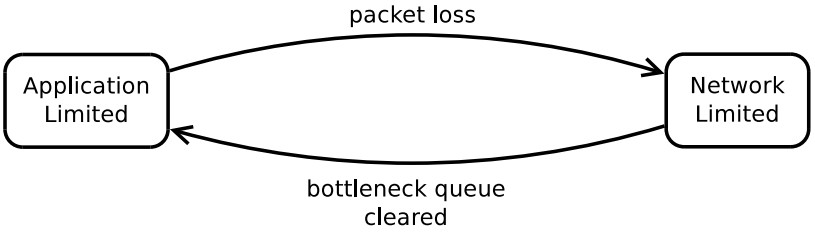


Figure 7.2: A TCP model for constant bitrate sources (adapted from Brosh, et al. [189]).

Application-limited and network-limited

We can characterize the state of TCP using a model like the one described in Figure 7.2, which was adapted from the model that Brosh, et al., used to study the delay friendliness of TCP for constant bitrate traffic [189]. According to such model, TCP can either be limited by the application or by the network.

There are two cases in which TCP may be limited by the application, namely, TCP may be limited either by the sender or by the receiver application. When TCP is limited by the sender application, it usually means that the sender is sending at a rate that is lower than the available bandwidth of the end-to-end path (which is, incidentally, the case that Brosh, et al., studied). In this condition, TCP should not create any queue over the end-to-end path, and there should only be occasional losses, caused by other sources that use (portions of) the same path. Another case in which

the sender is the limiting factor is when the default socket buffer size is smaller than the CWND, therefore the performance are limited by the buffer size. To solve this problem, Semke, et al., introduced the automatic socket send buffer tuning [282]. The second case in which TCP is limited by the application is the case in which the limiting factor is the receiver, i.e., when the RWND is smaller than the CWND. To avoid this performance penalty, modern operating systems automatically grow the RWND to keep it greater than the (estimated) sender's CWND [213].

On the contrary, when the application is sending at the maximum speed allowed by the network, the send buffer is large enough and the RWND is always lower than the CWND, we say that TCP is 'network limited' because its performance are determined by the temporal evolution of the CWND.

Of course, real TCP connections may experience application-limited periods followed by network-limited periods (as shown in Figure 7.2). As regards the transition from the application-limited to the network-limited states, when the limiting factor is the sender application the delay typically grows, because during the recovery the send rate is controlled by TCP (which sends as fast as possible) rather than by the application (which is imposing a certain rate). Conversely, when the limiting factor is the receiver's buffer, the delay should not grow because the sender can hardly sustain a high rate during the recovery: the receiver's buffer, in fact, typically ends up filled with the packets received after the lost one(s), therefore the receiver announces a zero-packet RWND; in turn, this fact, contributes to significantly reduce the network load and increases the likelyhood of a timeout. The same stall of the connection speed happens when, similarly, the limiting factor is the socket send buffer. As regards the transition from the network-limited to the application-limited states, the queuing delay tends to zero, because the send rate (whatever is the limiting factor) is lower than the available bandwidth, therefore no queue is created.

In our experiments we are not going to see cases in which the sender application limits the performance because our senders always try to send at maximum speed. However, we are going to use the model described in Figure 7.2 when we will discuss the few cases in which TCP was limited by either the socket send buffer or the RWND.

Base RTT

The 'base RTT' is the RTT measured when a TCP connection is established, which typically is also the minimum RTT (unless there are routing changes during the transfer). As we explain below, when the bufferbloat delay is large, the base RTT can be significantly smaller than the average RTT.

Bufferbloat

When a TCP connection (or more TCP connections) use all the available bandwidth, it creates a queue at the bottleneck. In turn, the queue at the bottleneck causes queuing delay, which adds to the base RTT of the connection(s). When the queuing delay skyrockets because the bottleneck is provisioned with huge buffers (compared

to its send speed), we have 'bufferbloat' and 'bufferbloat delays' [219]. Because home gateways are usually equipped with huge buffers, and because the uplink of many home networks is significantly slower than the WiFi network inside the home, the bufferbloat is often a problem that plagues home networks [203].

In particular, the bufferbloat was extremely annoying for BitTorrent users because BitTorrent is an application that aggressively uses the upload channel, thus creating significant bufferbloat delays. To solve this BitTorrent-specific problem, the BitTorrent developers recently proposed the LEDBAT protocol that we already mentioned in Section 7.1, which is designed to keep the bufferbloat delay under control. Recent studies documented that LEDBAT "alleviates the problem for most users" but "some users may still experience" bufferbloat delays as large as a few seconds [194].

Congestion window reduction

A 'Congestion Window Reduction' (CWR) happens when TCP reduces its CWND to respond to a congestion signal (i.e., a timeout, a triple-duplicate ACK, a SACK vector that lists more than three received packets, or other congestion signals[1]).

Outstanding window

Usually computed from a packet capture, the 'outstanding window' (OWND) is the estimate of the outstanding data (i.e., the data sent but not ACKed) and provides (in many cases) a good approximation of the CWND.

Pipeline

The end-to-end network between a TCP sender and a TCP receiver can be seen as a pipeline able to contain a maximum number of packets (which we called N in Section 7.2.2). In this sense, we will sometimes say that TCP 'fills the pipeline' or that 'TCP attempts to fill the pipeline' or that 'the pipeline is full'. In particular, saying that the pipeline is full is equivalent to say that TCP is using all the available bandwidth and is equivalent to say that TCP reached the equilibrium.

Recovery period

The 'recovery period' is the time interval from the receipt of the early congestion signal to the receipt of the first non-duplicate ACK that moves TCP from the RECOVERY state to the OPEN state.

The discussion now moves to the Inverse-Mathis model.

[1]For example, the explicit congestion notification (ECN) mechanism, described by RFC3168 [267], which we mention here to remind the reader that there are other (less-frequently used) types of congestion signals than the three congestion signals frequently mentioned in this Chapter.

7.3 The Inverse-Mathis Model

The first model for packet loss rate estimation at the application level presented in this doctoral thesis is called 'Inverse-Mathis model' (Inv-M). Such model is based on the well-known Mathis model that models the macroscopic behavior of the TCP congestion avoidance algorithm using a simple formula that predicts the download speed (henceforth, goodput) of a sustained, steady-state TCP flow [250].

We are, of course, well aware of the existence of TCP-performance models that are more advanced (and more complex) than the Mathis model. For example, in this work we could have used one of the following, more-advanced models: the PFTK model, which, unlike the Mathis model, also models timeouts and application-limited periods [261]; the model proposed by Cardwell, et al., which extends PFTK to model TCP latency and the effects of the slow start [191]; the model proposed by Guillemin, et al., which models congestion avoidance in presence of correlated losses that span multiple RTTs [221].

However, we decided to use the Mathis model, because its parameters (the goodput and the round trip time, henceforth RTT) are much more easily measurable at the application level than are the parameters used by other models.

7.3.1 Description of the Mathis Model

The simplicity of the Mathis model is made possibly by a number of assumptions; in the following, we list the ones that are more relevant to the Inverse-Mathis model:

1. the TCP flavor in use is Reno (or NewReno);
2. TCP updates its CWND using the congestion avoidance algorithm and reacts to (the few) early congestion signals using the fast-retransmit and the fast-recovery algorithms, i.e., no timeouts, no slow start;
3. SACKs are enabled;
4. TCP is network limited;
5. the RTT is constant over the path.

Under these assumptions the Mathis formula is:

$$goodput = \frac{MSS \cdot C}{RTT \cdot \sqrt{PLR}}, \tag{7.1}$$

where:

(a) *goodput* is the number of received bytes divided by the download time;

(b) MSS is the maximum segment size, which is typically 1460 bytes per packet, even if other sizes are possible;

(c) C is an empirical constant whose value incorporates both the loss model and the ACK strategy: when the loss model is random and TCP uses delayed ACKs[2], the value of C is 0.93 (see the Mathis paper [250] for a list of other possible values of C);

(d) RTT is the round trip time;

(e) PLR is the packet loss rate, i.e., the number of congestion signals divided by the number of packets that were sent.

7.3.2 Description of the Inverse-Mathis Model

Under assumptions 1-5 of Section 7.3.1, we invert the Mathis formula, to compute the PLR from the goodput and the RTT as follows:

$$EPLR = \left(\frac{MSS \cdot C}{goodput \cdot RTT} \right)^2 \tag{7.2}$$

Note that we called EPLR (for 'estimated PLR') the result of Equation 7.2, instead of just PLR, to underline that the result of Equation 7.2 is an estimate. To clarify the kind of PLR, in the following we will always call RPLR (for 'real PLR') the packet loss rate measured at network level.

In addition to the assumptions 1-5 of Section 7.3.1, in this work we also assume that:

6. the loss model is random and TCP implements delayed ACKs (i.e., $C = 0.93$);

7. the MSS is 1,460 bytes per packet.

Of course, different choices of the C and MSS parameters were possible. However, it is safe to assume that delayed ACKs are implemented, because they are indeed implemented by all the main operating systems. The assumption of random losses seems reasonable as well, at least for networks with moderate losses, because the SACKs (which are implemented by all the major operating systems) allow TCP to treat a few correlate losses that occur in the same RTT as a single loss[3]. As regards the MSS, in most cases the real MSS is either 1460 bytes or very close to 1460 bytes[4].

---

[2]That is, instead of ACKing every packet, TCP sends an ACK every two packets.

[3]This property of the SACKs is shown, for example, in the work of Mathis and Mahdavi that introduces the Forward-Acknowledgment (FACK) congestion-control algorithm [248] and that compares TCP FACK with TCP Reno and TCP Reno+SACKs.

[4]The most-common end-to-end path MTU (maximum transfer unit) is the Ethernet MTU of 1500 bytes (see pag. 55-57 of UNIX Network Programming [286]), therefore the MSS is 1460 bytes (1500 bytes minus 20 bytes for the IPv4 header minus 20 bytes for the TCP header). Values smaller than (but close to) 1460 are also very common and indicate the usage of network technologies that encapsulate the TCP/IP packet into a lower-level header (e.g., ADSL with Point-to-Point protocol over Ethernet encapsulation, which imposes a 1492 bytes MTU [243]). Accordingly, the most frequent MSS values that we observed in the wild (see Section 7.6.3 for more details) were the following: 1448 bytes (193,082; 34%), 1460 bytes (144,053; 25%), 1440 bytes (75,755; 13%), and 1452 bytes (60,675; 11%).

Before moving to the next Section, in which we discuss application-level measurements, let us make a final remark: as said the (Inverse-)Mathis model assumes that TCP is network limited, i.e., that the RPLR is always greater than zero. As a consequence, the PLR estimated by the Inverse-Mathis model could never be zero; therefore, the Inverse-Mathis model cannot detect cases in which there are no losses because, e.g., the TCP performance is limited by an under-dimensioned receiver's buffer.

7.3.3 Goodput and RTT Application-Level Measurements

The Inverse-Mathis model (Equation 7.2) is designed to estimate the PLR from simple application-level measurements (or reasonable estimates) of the goodput and of the RTT. In this Section we discuss how to measure the goodput and the RTT at the application level, and we discuss the accuracy of such measurements.

The goodput can be reliably measured at the receiver dividing the number of received bytes by the download time. Alternatively, one could also compute the goodput at the sender; however, the sender may underestimate the download time. The sender-side estimate of the download time, in fact, does not include the time required for emptying the socket send buffer (which may contain large amounts of data when the bandwidth-delay product, BDP, is large).

As regards the RTT, a simple upper-bound estimate of the (minimum) RTT can be obtained during the three-way handshake, by measuring the time elapsed in the connect() system call (from now on, 'connect delay'). This technique is unintrusive, and it can be easily implemented in existing applications; however, the connect delay does not capture subsequent RTT changes (including the bufferbloat delay).

To improve the robustness of the EPLR, therefore, the measurer should collect RTT estimates throughout the connection lifetime and then feed Equation 7.2 with the average RTT. However, this strategy is only feasible if the application and the application protocol allow the measurer to exchange small ping-like messages with the other endpoint[5].

However, for implementation simplicity, in this work we only used the connect delay, and we used the average RTT measured at packet level as ground truth. When we will use the average packet-level RTT as ground truth, we will often call the result B-IM EPLR (for 'Benchmark Inverse-Mathis EPLR') to clearly mark the difference between it and the EPLR estimated using the connect time. We plan to investigate more advanced RTT-estimation strategies as part of our future work.

[5]For example, an HTTP client could collect RTT samples during a long HTTP download as follows: after the download is started, the client will open a new HTTP connection with the same server, then it will periodically issue HEAD requests on such connection, thereby measuring the round-trip delay at the application-level (which is composed of the 'real' RTT plus the application-processing delay, a negligible time delay in many cases, unless the RTT is tiny and the server is trashing).

7.4 Evaluation of the Inverse-Mathis Model

The Inverse-Mathis model is evaluated using both a testbed environment and controlled Internet experiments. We use the testbed environment to evaluate the Inverse-Mathis model in a wide range of network conditions (i.e., varying RTTs and varying random PLRs); with controlled Internet experiments, instead, we evaluate the Inverse-Mathis model in Internet experiments in which we control at least one of the two endpoints (in the sense that we can capture packets, add more network losses, and/or inflate the RTT).

In particular, to emulate diverse network conditions (as well as to add more losses and/or inflate the RTT) we used Netem [98], a Linux kernel module that allows one to emulate the characteristics of wide area networks (WANs), by adding losses and/or delays to network interfaces. The basic Netem usage allows one to add a fixed delay and a random loss probability but more advanced usages are possible, allowing one to specify the delay distribution, the loss correlation, packet reordering, duplication and corruption.

To perform the experiments we used a piece of software written for the purpose, called TestbedKit [140]. This piece of software includes a simple TCP client and server, as well as scripts to control Netem and to periodically run tests. In TestbedKit goodput and RTT samples are collected at the application level: in particular, the RTT is estimated by measuring the connect delay and the goodput is measured at the receiver. The test duration is controlled by the sender, which stops sending after ten seconds and waits for the receiver to close the connection. We chose to run the test for ten seconds because we were looking forward to perform this kind of measurements using Neubot (in fact, the TestbedKit code was later adapted to Neubot and become the 'raw test'); therefore, ten seconds seemed to be a good compromise between accuracy and bandwidth usage, considering that Neubot is a tool running in the background. Also, ten seconds is the duration of the NDT download test [192].

In the following, we describe the evaluation of the Inv-M model using the testbed environment; next, we describe the results of controlled Internet experiments.

7.4.1 Evaluation Using a Testbed

In this Section we describe the testbed experiments. The testbed was composed of three Linux 3.0.0 virtual machines running on VirtualBox and connected using the 'internal network emulation' feature of VirtualBox. The sender and the receiver machines were attached to two diverse virtual networks. In turn, such two virtual networks were interconnected by the third machine, which ran Netem and routed the packets. Overall, when Netem did not kick in, we could transfer more than 300 Mb/s with negligible RTT overhead (less than a few milliseconds). We divide the discussion of the testbed experiments results in two pieces: at first, we evaluate the Inverse-Mathis model at the application-level only, then we analyze a few packet traces to gain further insights into the model behavior.

Application-Level Results In this paragraph we discuss the application-level results of the set of experiments called 'VTestbed' (for 'virtual testbed'). In such set of experiments

IPLR	IRTT = 20 ms	40 ms	60 ms	...	280 ms	300 ms
$3.0 \cdot 10^{-2}$	1–25	26–41	42–57	...	418–233	234–249
...	...	...	...	...	...	...
$2.0 \cdot 10^{-2}$	1210–1225	1226–1241	1242–1257	...	1418–1433	1434–1449
...	...	...	...	...	...	...
$1.0 \cdot 10^{-2}$	2410–2425	2426–2441	2442–2457	...	2618–2633	2634–2649
$8.0 \cdot 10^{-3}$	2650–2665	2666–2681	2682–2697	...	2858–2873	2874–2889
...	...	...	...	...	...	...
$1.0 \cdot 10^{-3}$	3610–3625	3626–3641	3642–3657	...	3818–3833	3834–3849
$8.0 \cdot 10^{-4}$	3850–3865	3866–3881	3882–3897	...	4058–4073	4074–4089
...	...	...	...	...	...	...
$1.0 \cdot 10^{-4}$	4810–4825	4826–4841	4842–4857	...	5018–5033	5034–5049
$8.0 \cdot 10^{-5}$	5050–5065	5066–5081	5082–5097	...	5258–5273	5274–5289
...	...	...	...	...	...	...
$1.0 \cdot 10^{-5}$	6010–6025	6026–6041	6042–6057	...	6218–6233	6234–6249

Table 7.1: VTestbed experiments. The rows indicate the IPLR values, and the columns indicate the IRTT values. At the intersection of a row and a column, we indicate the experiment ID corresponding to a given IPLR and IRTT. The IDs are, in turn, line numbers in the CSV file that contains the results.

we used Netem to vary the Netem-imposed RTT (henceforth, IRTT) and the Netem-imposed PLR (henceforth, IPLR) as described in Table 7.1, which is organized as follows.

The rows of Table 7.1 indicate the diverse values of the IPLR, and the columns indicate the diverse values of the IRTT. At the intersection of a row and a column we indicate the IDs of the experiments run with a given IPLR and a given IRTT. Because the results are saved in a CSV file, in particular, the experiment ID indicated in Table 7.1 is the experiment line number in the CSV file. For example, the results of the experiments run with IRTT = 20 ms and IPLR = $2.0 \cdot 10^{-2}$ occupy the lines 1210–1225 of the CSV file.

Note that, to fit Table 7.1 into a single page, we omitted a number of rows and columns from it. However one can easily compute the missing rows and columns, knowing that:

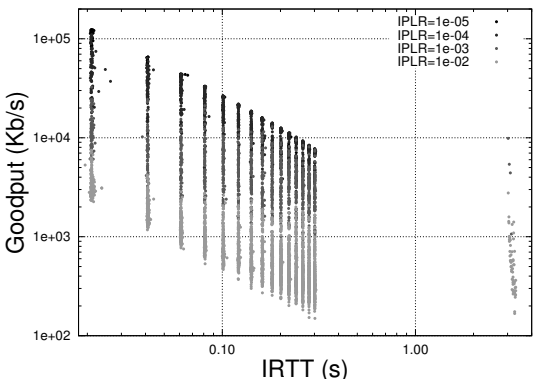
1. the full table has 15 columns, and the RTT increment between a column and the subsequent column is 20 ms;

2. we omitted the rows corresponding to $2.8 \cdot 10^{-2}$, $2.6 \cdot 10^{-2}$, $2.4 \cdot 10^{-2}$, $2.2 \cdot 10^{-2}$, $1.8 \cdot 10^{-2}$, $1.6 \cdot 10^{-2}$, $1.4 \cdot 10^{-2}$, $1.2 \cdot 10^{-2}$, $6.0 \cdot 10^{-3}$, $4.0 \cdot 10^{-3}$, $2.0 \cdot 10^{-3}$, $6.0 \cdot 10^{-4}$, $4.0 \cdot 10^{-4}$, $2.0 \cdot 10^{-4}$, $6.0 \cdot 10^{-5}$, $4.0 \cdot 10^{-5}$, and $2.0 \cdot 10^{-5}$;

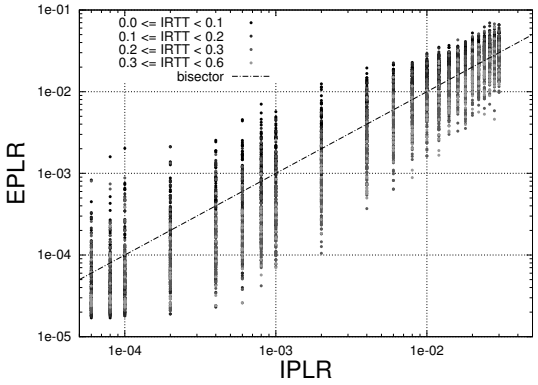
3. in general, we repeated each experiment 16 times, except that we repeated 25 times the experiment in which RTT = 20 ms and IPLR = $3 \cdot 10^{-2}$.

Let us now discuss the application-level results of this set of experiments. We start by illustrating the correlation between the goodput and the IRTT, then we study the correlation between the IPLR and the EPLR.

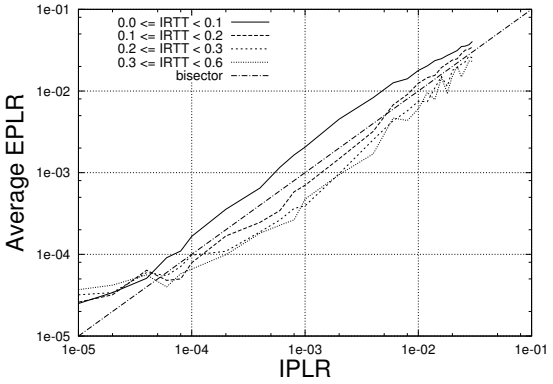
Figure 7.3a shows the goodput as a function of the IRTT. Each point is the result of a single TCP download experiment run using TestbedKit (henceforth, 'TestbedKit experiment') under a given IPLR and a given IRTT. Points with different IPLRs are colored using different levels of gray, to show more clearly that (according to Equation 7.1) the



(a) Goodput vs IRTT



(b) EPLR vs IPLR



(c) Average EPLR vs IPLR

Figure 7.3: VTestbed experiments results. IRTT and EPLR are, respectively, the Netem-imposed RTT (round trip time) and the Netem-imposed PLR (packet loss rate). EPLR is the PLR estimated with Equation 7.2.

higher the IPLR is, the lower the goodput is. The outlier points at the bottom right of the Figure are caused by lost SYN (or SYN|ACK) segments; in accordance with the Mellia-Zhang model [253], in fact, their connect time is three seconds (the default RTO value) plus one RTT. To produce clearer plots, these outliers have been omitted in subsequent plots (and, of course, they were not used to compute the EPLR).

Figure 7.3b shows the EPLR (computed using Equation 7.2) as a function of the IPLR. Each point is the result of a single TestbedKit experiment and points with diverse RTTs are colored with different levels of gray. Although there is a lot of variation, the data pattern suggests that points with smaller IRTTs are more likely grouped at the top of the plot; similarly, the data pattern suggests that points with higher IRTTs tend to group at the bottom. To better investigate this behavior, we averaged points having the same IPLR and the same IRTT order of magnitude.

The averages are shown in Figure 7.3c, which leads to the following observations:

(i) the curves loosely follow the bisector line, i.e., the line that represents the exact prediction by a model that covers the whole IPLR range;

(ii) the average of points with IRTT lower than 0.1 s is above the bisector line, the other averages are below the bisector;

(iii) the model is more accurate inside the $[10^{-4}, 10^{-2}]$ IPLR region, and less accurate outside of such region: in particular, the curves tend to flatten at low IPLRs (below 10^{-4}), moreover they have some oscillations at very high IPLRs (above 10^{-2}).

We note with moderate satisfaction that the EPLR curves loosely follow the bisector line; however, to move forward the discussion of the Inv-M model as a general tool to predict the PLR from application-level measurements, we need to explain the dependency on the IRTT (observation ii) and we need to explain why the model is more accurate within the $[10^{-4}, 10^{-2}]$ region (observation iii).

As regards observation ii), it is natural to make the working hypothesis that the dependency on the IRTT is caused by the sender's TCP flavor. As mentioned in Section 7.2.3, in fact, Linux uses by default TCP Cubic, a TCP variant whose average CWND is comparable with the NewReno CWND for low RTTs (and/or BDPs) and larger otherwise.

As regards the flattening curves at low IPLRs (observation iii), a tentative explanation is the following: assuming that MSS and C are adequate for our setup, Equation 7.2 says that we need to underestimate either the goodput or the RTT to obtain an EPLR higher than the IPLR and, hence, a flattening curve. Because the goodput measurement is straightforward, our initial hypothesis is that our RTT estimate underestimates the real RTT, possibly because our estimate (the connect time, which is measured before the transfer) does not account for the bufferbloat delay at the bottleneck.

As regards the EPLR oscillations at very high IPLRs (observation iii), there can be many explanations of this phenomenon. Our tentative explanation is that, at very high IPLRs, the variance of the goodput is high because TCP may not always manage to recover from the many losses without avoiding a timeout. In turn, with higher variance, one needs more samples to compute a robust average, otherwise the average may oscillate depending on the casual prevalence of lossy or lossless runs in each set of experiments.

To verify all the above working hypotheses we performed testbed experiments in which we also captured the packets to obtain a ground truth. We describe such new experiments and the related results in the following paragraph.

Packet-Level Analysis In this paragraph we describe the results of the experiments called 'VTestbed2u' (for 'virtual testbed #2 using Tcpdump'). VTestbed2u is equal to VTesbed, except that we also captured packets at the sender using Tcpdump [138].

Table 7.2 describes how we varied the IPLR and the IRTT. The table rows correspond to varying IRTT, and the table columns correspond to varying IPLR; moreover at the intersection of a row and of a column we indicate the IDs of the experiments run with a given IPLR and a given IRTT. As before, the experiments IDs are the row numbers in the

RTT	IPLR = $3 \cdot 10^{-2}$	$2 \cdot 10^{-2}$	$1 \cdot 10^{-2}$	$1 \cdot 10^{-3}$	$1 \cdot 10^{-4}$	$1 \cdot 10^{-5}$
20 ms	1–4 96–99	17–20 112–115	33–36 128–131	49–52 144–147	65–68 160–163	81–84 176–179
100 ms	5–8 100–103	21–24 116–119	37–40 132–135	53–56 148–151	69–72 164–167	85–88 180–183
200 ms	9–12 104–107	25–28 120–123	41–44 136–139	57–60 152–155	73–76 168–171	89–92 184–187
300 ms	13–16 108–111	29–32 124–127	45–48 140–143	61–64 156–159	77–80 172–175	93–96 188–191

Table 7.2: VTestbed2u experiments. The rows indicate the IRTT values, and the columns indicate the IPLR values. At the intersection of a row and a column we indicate the experiment IDs that correspond to a given IPLR and IRTT. The IDs are line numbers in the CSV file that contains the VTestbed2u experiments results, moreover the IDs are also indexes of the packet traces within the capture file.

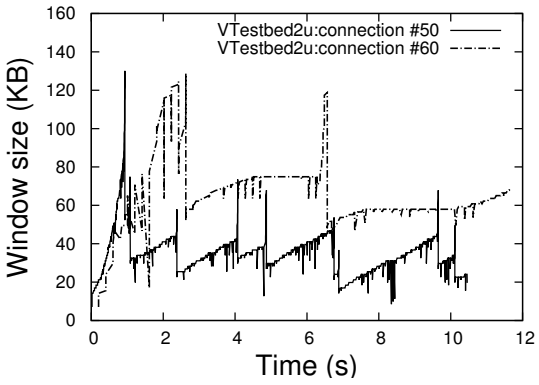
CSV file that contains the application-level results, moreover the IDs are also the indexes of the corresponding packet traces in the packet capture file; i.e., the application-level results that are in row four of the CSV file correspond to the fourth packet trace in the capture file.

Let us now describe how we analyzed the captured packet traces: the main tool that we used is Tcptrace [139], a packet-trace processing tool that performs many types of TCP-level analysis. In particular, we used Tcptrace to count the number of retransmissions that occur during the lifetime of a TCP connection, as well as to obtain the number of sent packets, the minimum and the average RTT. Moreover, we also used Tcptrace to produce OWND plots.

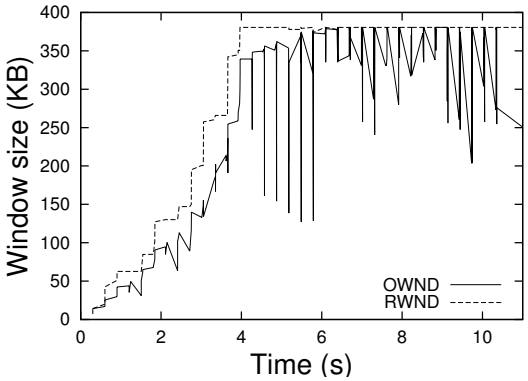
Returning to the observations that we made in the previous paragraph, the first phenomena that we want to investigate at packet level is the correlation between the EPLR and the IRTT. At the end of the previous paragraph, we made the hypothesis that this correlation can be caused by TCP Cubic. To understand the impact of TCP Cubic on the Inv-M model predictions, we manually inspected a number of OWND plots, two of which (connection #50 and connection #60) are shown in Figure 7.4a. We can easily see that the OWND of connection #60 (which has a large RTT) follows the typical Cubic TCP pattern (concave growth, plateau, convex growth, loss). On the contrary, connection #50 (which has a small RTT) exhibits a NewReno-like sawtooth pattern, probably because of the Cubic TCP-friendliness mechanism. Moreover, in accordance with the Cubic authors' simulations [222], at large RTTs the OWND is larger than at small RTTs. From what we empirically noticed we can draw the following conclusions:

1. thanks to the Cubic TCP-friendliness mechanism, when the RTT is small-to-medium (below, say, 100 ms) the Inverse-Mathis model should estimate the PLR in a reasonably accurate way, for both NewReno and Cubic;

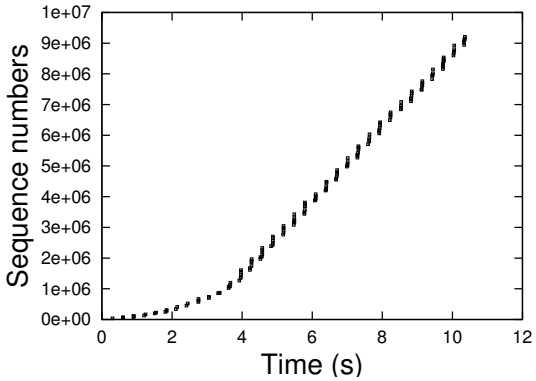
2. at large RTTs Cubic should be faster than NewReno, therefore the Cubic EPLR should be lower than the EPLR computed using Equation 7.2 (which allows us to fully explain observation ii);



(a) OWND of connections #50 and #60



(b) OWND of connection #95



(c) Time-sequence graph of connection #95

Figure 7.4: VTestbed2u experiments results. OWND is the outstanding window, a proxy of the congestion window (CWND) computed by Tcptrace from the packet capture. RWND is the receiver-advertised window, i.e., the buffer space available at the receiver.

3. it appears that we should run a similar set of testbed experiments to extensively cover NewReno; however, as we will see in the following, we already have plenty of Inv-M NewReno data because M-Lab uses NewReno by default;

4. it should be possible to derive an Inverse-Mathis-like formula specifically tailored for the case in which the sender uses Cubic, using as starting point, e.g., the average Cubic CWND formula proposed by Poojary and Sharma [263].

The second phenomena that we want to investigate concerns the flattening curves at low IPLRs. Our working hypothesis for explaining this phenomenon was the queuing delay; however the packet-level analysis does not confirm such hypothesis. There is, in fact, virtually no RTT variation during the transfer (in many cases the RTT varies no more than 2-3 millisecond, which is a very contained increase). Therefore the explanation of the flatting curves must be another one, namely, as it turns out, the absence of losses.

Figure 7.4b shows the OWND and the RWND of connection #95: after a long slow start, the connection performance is RWND-limited. Note that the oscillations, in particular, are

not CWRs, because the CWND after a CWR is expected to be lower than it was before the CWR. On the contrary, the oscillations in Figure 7.4b depend on the RWND-limited connection: because the pipeline has room for more packets than the RWND allows, TCP sends bursts of packets that fill the RWND, followed by silence periods in which it waits for the receiver to advertise that more space is available. This explanation is confirmed by the time-sequence-graph (Figure 7.4c) that shows bursts followed indeed by silence periods.

To conclude our discussion of the flattening curves at low IPLRs: we have shown that at low IPLRs there are very few (if any) loss events; however, as seen in Section 7.3.2, the Inverse-Mathis model always assumes that there are losses, therefore it always yields a PLR that in absence of losses is a 'virtual' PLR related to the RWND and to the RTT. This observation suggests that it is more robust to disregard the Inverse-Mathis model predictions at low IPLRs, given that Inv-M cannot detect the absence of losses.

As regards the observation that at very high IPLRs the average curves oscillate, we postulated that such phenomenon was caused by higher goodput variation than at other IPLRs, coupled with under-sampling of the channel. That is, we assumed that there are oscillations because sixteen repetitions are not enough to correctly estimate the frequency of lower-goodput runs in very high IPLR experiments. In turn, we assumed that the goodput variation was caused by timeouts provoked by the very high loss rate.

The packet-level analysis does not reveal, however, the presence of timeouts. We noticed, instead, that the distance between the losses was reduced at very high IPLRs, leading to more SACKs-triggered retransmissions per recovery period. For example, Tcptrace shows that, for IPLR equal to $3 \cdot 10^{-2}$ and RTT equal to 300 ms, TCP retransmits multiple packets per recovery period in five tests (#13–#15, #109 and #112) out of eight. Accordingly, the average recovery period is longer than the RTT, e.g., in test #13 the average recovery period is 486.9 ms (compared to a RTT of 300 ms). As a consequence, the goodput should be lower than the goodput of a connection with shorter recovery periods.

The longer-recovery-period phenomenon that we unveiled provides insight, however it does not explain the phenomenon of the average-curves oscillations. The link between the longer recovery period and a significant reduction of the goodput, in fact, is still a working hypothesis for us, but we look forward to further explore this subject in the future. For now, let us conclude our discussion of the oscillating curves by saying what we know for sure: the Mathis model assumes that losses are spaced enough for TCP to recover in a single RTT, therefore we would not be surprised to find Inv-M less reliable at very high IPLRs.

Concluding Remarks We end this Section devoted to testbed experiments with the positive note that the Inv-M EPLR is quite close to the IPLR in many cases. However, we also remark that the EPLR is correlated with the IRTT at large IRTT when the Cubic TCP flavor is used (instead of the traditional NewReno flavor). On the contrary, when the IRTT is medium or small and Cubic is used, Inv-M is more reliable because Cubic has a TCP-friendliness mechanism that guarantees NewReno-like behavior in such condition. We also defined more clearly when the Inv-M model is supposed to work reliably, concluding

that Inv-M is not reliable at low IPLRs because it cannot detect the case in which there are no losses. We also note that the average EPLRs oscillate at very high IPLR, but we could not find a convincing packet-level explanation of such phenomenon.

7.4.2 Evaluation Using Controlled Internet Experiments

In this Section we describe the evaluation of the Inverse-Mathis (Inv-M) model using ADSL-and-WAN controlled Internet experiments. Unlike testbed experiments, the experiments that we present in this Section are performed in a more realistic scenario in which the traffic always traverses a portion of the Internet. We say that the experiments described in this Section are 'controlled' in the sense that we always controlled at least one of the two involved endpoints, which allowed us to capture packets using Tcpdump as well as to use Netem to add random losses and/or to increase the base RTT.

This Section is organized as follows: we start by providing a description of the experiments; next, we explain how we measured the packet-loss rate (PLR) at the packet level; then, we show the experiments results; finally, we draw the conclusions.

Controlled-Internet-Experiments Description Table 7.3 shows an overview of the diverse experiments performed. In short, we varied the access network technology, the Netem-imposed packet loss rate (henceforth, IPLR), the Netem-imposed RTT (henceforth, IRTT), the operating system, and the congestion avoidance algorithm. More details regarding each experiment are given below.

	ADSL #1	ADSL #2	ADSL #3	WAN #1	WAN #2
Bandwidth	7 Mb/s		512 kbit/s	100 Mb/s	
Base RTT	60 ms			18 ms	51 ms
Sender	Linux 3.0.0	OpenBSD 5.5-current	MacOS 10.8.5	Linux 3.0.0	
CongAvoid	Cubic	NewReno		Cubic	
Router	Linux 3.0.0	Linux 3.10.17	N/A	Linux 3.0.0	
Receiver	MacOS 10.6	Linux 3.2.0	Linux 3.10.17	Linux 2.6.32	
Netem Used	Yes		No	Yes	
IPLR range	[0, 0.005]		N/A	[0, 0.0025]	[0.002, 0.002]
IRTT	N/A				0.033 s

Table 7.3: Controlled Internet experiments. We varied the access network technology, the Netem-imposed packet loss rate (IPLR), the Netem-imposed RTT (IRTT), the operating system, and the congestion avoidance algorithm.

The first two experiments that we performed, the ADSL #1 and the WAN #1 experiments, were designed to assess the accuracy of the Inverse-Mathis model, both in presence and in absence of random losses, with two common access-network technologies.

In the ADSL #1 experiment we performed a number of downloads over an end-to-end channel with 7 Mb/s of (nominal) downstream bandwidth and 60 ms of average base RTT (measured with the ping tool). The sender, a Linux 3.0.0 netbook using TCP Cubic, was attached to a router, implemented by a Linux 3.0.0 workstation that was part of our campus LAN. The receiver was a MacBook Pro running MacOS 10.6 and connected to

the ADSL through a 11 Mb/s WiFi network. During the experiment we used Netem (on the workstation) to impose the following random IPLRs: zero[6], 0.001, 0.002, 0.003, 0.004, 0.005. For each value of the IPLR we performed sixteen downloads, in our experience a number of repetitions sufficient to account for transient anomalies.

In the WAN #1 experiment we performed a number of downloads over an end-to-end channel with 100 Mb/s of (nominal) downstream bandwidth and 18 ms of average base RTT (measured with the ping tool). The sender, a Linux 3.0.0 netbook using TCP Cubic, was attached to a router, implemented by a Linux 3.0.0 workstation that was part of our campus LAN. The receiver was a Linux 2.6.32 OpenVZ virtual-private server hosted by a well-provisioned ISP. During the experiment we used Netem (on the workstation) to impose the following random IPLRs: zero, $5 \cdot 10^{-5}$, 10^{-4}, $2.5 \cdot 10^{-4}$, $7.5 \cdot 10^{-4}$, 0.0025. For each value of the IPLR we performed sixteen downloads.

Next, we performed the WAN #2 experiment to check whether the Inv-M EPLR was equal for diverse access-network technologies in similar conditions. To this end, we imposed to the WAN connection conditions (base RTT = 51 ms, IPLR = 0.002) similar to the ones experienced by one of the ADSL #1 experiments (base RTT = 0.06 s, IPLR = 0.0025). Because the base RTT of the WAN experiment was 0.018 s, in the WAN #2 experiment we added 0.033 s of IRTT using Netem. We repeated the download sixteen times.

Successively, we performed the ADSL #2 experiment, in which we replaced the Linux sender (using Cubic) with an OpenBSD sender (using NewReno), to study the Inv-M accuracy with NewReno. During the experiment we used Netem (on the workstation) to impose the following random IPLRs: zero, 0.001, 0.002, 0.005. For each value of the IPLR we performed sixteen downloads.

The last experiment, ADSL #3, was constructed to show the impact of the bufferbloat delay on the EPLR. Unlike the other experiments, in this case: we used the (slower) ADSL upload channel, because a slow channel allows one to create bufferbloat delay more easily; we did not inject random losses because random losses help to shorten the queues, mitigating the bufferbloat delay; we ran a single 120 s experiment because it took 50 seconds to fill the router's buffer. In this experiment: the sender was a NewReno MacOS 10.8.5 MacMini, attached to the ADSL network of the ADSL #1 and #2 experiments; the receiver was a Linux 3.10.17 workstation located in our campus LAN.

Packet-Level PLR Measurements In testbed experiments, the PLR at packet level was not measured: we limited ourselves to show that the EPLR curves loosely follow the bisector line. We took such approach because we assumed that all the losses in the testbed were caused by Netem. However, such approach is no longer possible in this Section: since we comment on experiments that involve a portion of the Internet, in fact, it is unlikely that all the losses are caused by Netem. Hence, we need to define a way to measure the losses from the captured packet traces.

[6]When the IPLR was zero, the TCP connection still suffered from the losses caused by the queue of the bottleneck, which in this case was the 7 Mb/s ADSL connection.

Intuitively, the packet-level PLR is defined as the number of losses divided by the number of sent packets. However, this definition of the packet-level PLR is not completely adequate for the needs of the Mathis model. In Section 7.3.1, in fact, we said that for the Mathis model the PLR is the number of congestion signals divided by the number of sent packets. Therefore, as long as there is one loss per congestion signal, the intuitive packet-level PLR definition is a reasonable proxy of the PLR expected by the Mathis model; on the contrary, when losses are correlated (so that many losses occur in a short time frame and correspond to a single congestion signal), the intuitive definition is less accurate.

Unfortunately, Tcptrace allows one to count the number of losses very easily (it suffices to use the number of retransmitted packets) but, on the contrary, the number of congestion signals can only be obtained indirectly from the Tcptrace OWND plot. By looking at such plot, a trained human can count the number of CWRs and, hence, the number of congestion signals. However, this process is slow, error prone, and boring.

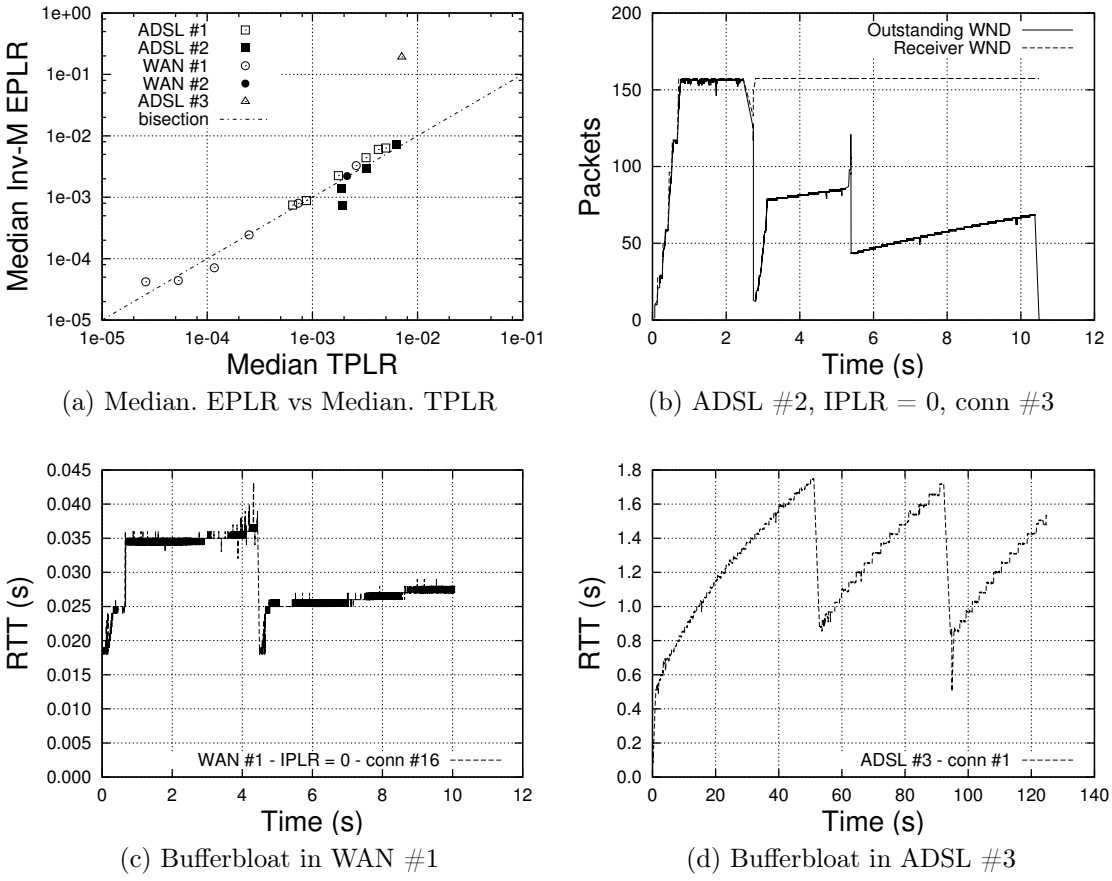
In the end, for simplicity we chose to compute the packet-level PLR as the number of retransmitted packets divided by the number of sent packets. However, because we know that such number is not always a good approximation of the CWRs, we also counted the number of CWRs when we suspected that losses were highly correlated. Also, for the same reason, in the following we are not going to call this packet loss rate RPLR, because it is not a real PLR; rather we will call it TPLR (for 'Tcptrace PLR').

Internet Experiments Results Figure 7.5a shows the median Inv-M EPLR in function of the median TPLR. In this Section we chose to use the median, rather than the average, because the median is much more robust to outliers than the average, and because there are a few outliers among the results of the Internet experiments. In most cases the median was computed on the sixteen results of TestbedKit experiments performed in similar network conditions, with the exception of the ADSL #3 point, which is the result of a single TestbedKit experiment. In general, we can say that the Inv-M model predicts quite reliably the TPLR; more details in the following.

As regards the ADSL #1 experiment, in which Cubic TCP was used, the medians align very closely to the bisector line. In all cases the EPLR and the RPLR have the same order of magnitude. The biggest relative error, 0.42, occurs for the set of points in which the Netem-imposed packet loss rate was 0.004.

As regards the ADSL #2 experiment[7], in which NewReno TCP was used, points align near the bisector for TPLRs higher than $2 \cdot 10^{-3}$; below that threshold, the TPLR does not change significantly. Guessing that this phenomenon is caused by correlated losses, we manually inspected the OWND plot, shown in Figure 7.5b. This plot tells us that

[7]In this experiment the connect time also included the DNS request time, because of a TestbedKit software bug. Later we noticed the same bug in WAN #1 and WAN #2 experiments. Therefore, in all these experiments we computed the EPLR using the minimum RTT provided by Tcptrace rather than the TestbedKit connect time. The methodological legitimacy of this step is confirmed by subsequent connect-only experiments, performed after we fixed the bug, in which the TestbedKit connect time is found to be equal to the minimum Tcptrace RTT plus a small amount of random noise.



(a) Median. EPLR vs Median. TPLR

(b) ADSL #2, IPLR = 0, conn #3

(c) Bufferbloat in WAN #1

(d) Bufferbloat in ADSL #3

Figure 7.5: Internet experiments results, showing that the Inv-M model reliably predicts the median Tcptrace Packet Loss Rate (TPLR) in most experiments, except the ADSL #3 experiment (Figure 7.5a); that there are highly correlated losses in the ADSL #2 experiment (Figure 7.5b); that there is high queuing delay both in the WAN #1 and in the ADSL #3 experiments (respectively Figure 7.5c and 7.5d).

the RWND limits the OWND at the beginning of the experiment (probably reducing the likelyhood of a successful recovery); later the connection experiences a single CWR in which many packets are lost and, in a subsequent CWR event, the connection loses other packets as well. To confirm that the behavior of the ADSL #2 points depends on the correlated losses, we compute the RPLR using the number of CWRs: the connection shown in Figure 7.5b has EPLR $= 7 \cdot 10^{-4}$, sent 6019 packets, lost eleven packets, and experienced two CWRs; therefore, using the number of lost packets, $TPLR = 11/6019 = 1.8 \cdot 10^{-3}$, however, using the number of CWRs, $RPLR = 2/6019 = 3.3 \cdot 10^{-4}$.

As regards the WAN #1 experiment, in which Cubic TCP was used, points align close to the bisector line. For TPLR equal to 10^{-4} losses are more correlated than they are for high PLRs, therefore the TPLR is higher than the RPLR (as happened for the ADSL #2 experiment). For TPLRs lower than 10^{-4}, instead, the median points return close to

the bisector line; also, notably, the point with minimum TPLR is above the bisector. The latter phenomenon is caused by the bufferbloat delay; the point with minimum TPLR, in fact, is the median of all the experiments performed with IPLR = 0, in which the TCP connection used all the available bandwidth. In turn, because all the bandwidth was used, the queue on the bottleneck grew, creating additional queuing delay. This explanation is confirmed by Figure 7.5c, in which we see the RTT computed by Tcptrace in function of the time for connection #16 and IPLR = 0. The RTT, in fact, follows a CWND-like pattern: in particular, before the first loss (which, of course, reduces the queuing delay), the RTT is about two times the initial RTT. As a consequence, the EPLR computed using the average RTT is (only slightly) more accurate than the one computed using the initial RTT: connection #16 sent 76840 packets, nine of which were lost, moreover, the connection experienced two CWRs; also, the connect time RTT (measured with Tcptrace, as explained in note) was 18.4 ms, and the average RTT was 30.0 ms; with the above numbers we obtain the following results: $RPLR = 2/76840 = 2.6 \cdot 10^{-5}$, EPLR (using the connect time) = $4.1 \cdot 10^{-5}$, and EPLR' (using the average RTT) = $1.56 \cdot 10^{-5}$.

The impact of the bufferbloat on the Inverse-Mathis EPLR computed using the connect time is confirmed by the results of the ADSL #3 experiment, in which we performed an experiment designed to maximize the bufferbloat. The plot in Figure 7.5d, which shows the RTT in function of the elapsed time, confirms that the experiment generated a huge bufferbloat delay: the RTT, which follows a CWND-like pattern, grows as high as 1.7 seconds before a loss contributes to reduce the queuing delay. The magnitude of the RTT explains why the ADSL-#3 point in Figure 7.5a (TPLR = $7.1 \cdot 10^{-3}$, EPLR = $1.9 \cdot 10^{-1}$) is an outlier: the connect time is too small compared to the average RTT. Conversely, we obtain a less imprecise estimate ($4.7 \cdot 10^{-4}$) if we use the average RTT.

As regards the WAN #2 experiment, in which Cubic TCP was used, the median of the sixteen experiments is on the bisector line of Figure 7.5a. In this experiment we imposed to the WAN connection a PLR and a RTT similar to the ones imposed during one run of the ADSL #1 experiment. We did that to understand the link between the different bandwidth (100 Mb/s vs. 7 Mb/s) and the EPLR. The comparison of this experiment to the similar ADSL experiment suggests that, when not all the bandwidth is used and when there are enough random losses to avoid any bufferbloat, diverse access network technologies have similar EPLRs in similar conditions (i.e., RTT and IPLR).

Concluding Remarks Through controlled Internet experiments we showed that the Inverse-Mathis model is quite reliable to estimate the losses of connections having short RTTs, both with NewReno and with Cubic. We identified a couple of cases in which losses were too correlated for our (simple) packet-level PLR estimate to be good approximation of the number of congestion signals divided by the number of sent packets (which is the Mathis-model definition of PLR). In such cases we showed that the EPLR is more accurate if we compute the PLR as expected by the Mathis model (in the following Sections we will avoid this problem using the number of congestion signals exported by the Web100 TCP/IP stack [251] installed on all M-Lab [206] servers). We also discussed the impact

of the bufferbloat on the PLR estimated by the Inv-M model. As we showed through examples, when the bufferbloat delay is high we cannot trust the EPLR computed using the connect time, which can be as large as 100 times the packet-level PLR.

7.5 The Likely-Rexmit Model

We designed the Likely Rexmit (L-Rex) model to be more accurate than Inv-M in two key scenarios in which Inv-M accuracy is unsatisfactory: when there is significant bufferbloat delay and when there are no losses. Moreover, being L-Rex based on the empirical observation of the receiver-side application-level effect of the fast retransmit algorithm, its accuracy should be independent of the TCP flavor in use, because the fast retransmit is a common feature of most (if not all) TCP flavors. However, as presented in this doctoral thesis, L-Rex may not be equally accurate in all access networks, because we specifically optimized its parameters for ADSL and Fast Ethernet access network, which are (with cable) the most commonly-used landline Internet connections[8].

The General Idea The model is called 'Likely Rexmit' because it analyzes the application-level dynamics of the recv() system call to identify likely 'rexmit' (i.e., retransmission) events. To better understand L-Rex, it is useful to describe how we designed it. We started from the empirical observation of what happens at the application receiver side of ADSL and Fast Ethernet connections subject to moderate losses:

1. since recv() returns as soon as data is available, it is usually triggered at the reception of segments. Typically, in slow networks (e.g., ADSL) one segment is received at a time, while faster networks (e.g., Fast Ethernet) receive one or two segments with similar probability. In short, any level-two access network has its typical number of segments and, hence, of bytes returned by recv();

2. during a burst of packets, the time elapsed between the occurrence of two consecutive recv()s is very small, because many back-to-back packets are received;

3. between two consecutive bursts there is typically a 'silence period', which is usually smaller than one RTT. If the pipeline is full, however, the receiver sees the continuous arrival of more-or-less equally-spaced segments (i.e., there is no silence period);

Successively, we observed what happens after a loss:

4. the silence period is longer than usual (typically more than one RTT). In fact, recv() cannot return because it is waiting for the missing segment(s). In the best case (i.e.,

[8]ADSL and Fast Ethernet were chosen because they represent important Internet-access technologies to which we had easy access. It should be possible, of course, to optimize L-Rex for other access networks: to do so, it suffices to find out the optimal value of the L-Rex parameters for such access networks.

fast retransmit) an additional RTT is needed to retransmit the packet and to trigger recv(), but in many cases the recovery can take more than one RTT;

5. moreover, after a recovery, recv() very often returns a large, non-typical number of bytes (the lost segment plus all the segments arrived while recv() was blocked).

We empirically observed that many losses followed the pattern described by 4. and 5.: a longer-than-usual silence period, and a large, non-typical number of bytes. We used this information to define the algorithm described in the following paragraph.

The L-Rex Algorithm The L-Rex algorithm is based on two empirical rules that, in turn, are based on the empirical observations described above. As we will see, the two empirical rules allow one to compute the number of likely retransmissions, which is then used to compute the estimated packet loss rate (EPLR).

The two empirical rules operate on a list that contains the number of bytes received by each recv(), called RRB for 'recv()-received bytes', and the time elapsed since previous recv(), called IRG for 'inter-recv() gap'. As a preliminary step, before we can apply the rules, we need to scan the list once, to compute the RRB frequency. Then, we scan again the list and we count the number of cases for which:

(i) the IRG is greater than $H \cdot RTT$, where RTT is the connect time, and H is an empirical parameter described below;

(ii) the RRB is (a) greater than 1 MSS (Maximum Segment Size) and (b) less frequent than K, where K is an empirical parameter described below.

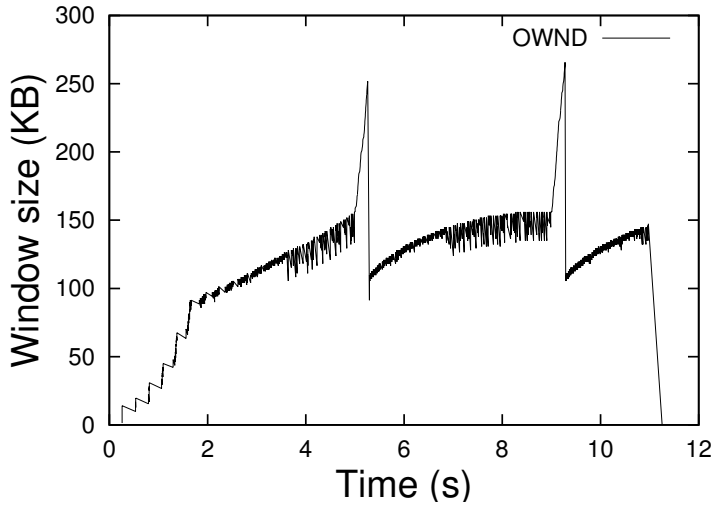
Once the number of estimated losses is known, the L-Rex estimated PLR (L-Rex EPLR) is computed using the following formula:

$$EPLR_{L-Rex} = \frac{losses}{total\_bytes \ / \ MSS} \tag{7.3}$$

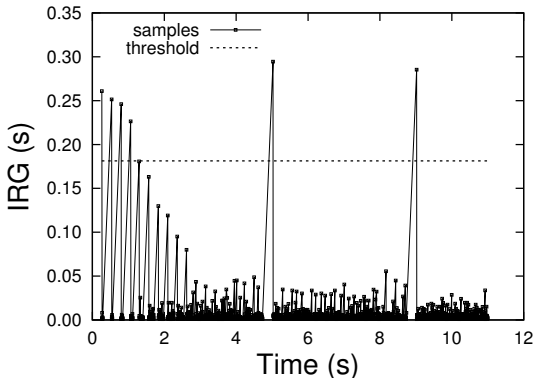
where *losses* is the number of estimated losses, *total_ bytes* is the total number of received bytes, and MSS is the maximum segment size.

Running repeated experiments, we optimized the value of the parameters H and K to maximize the number of identified losses for ADSL and Fast Ethernet access networks (chosen because they represent important Internet-access technologies to which we had easy access) with moderate losses. The optimal values are, respectively, 0.7 and 1%. The optimal value of H is lower than one for robustness, because we observed that the connect time RTT estimate is higher than the real minimum RTT in some cases.

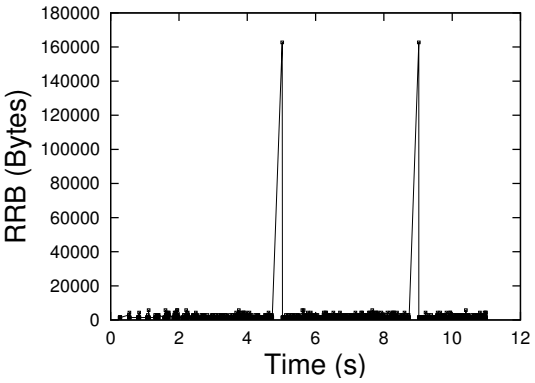
Unlike Inverse Mathis EPLR estimates, the L-Rex EPLR can be zero. Of course, however, there is no certainty that a zero value corresponds to no losses. For example, when losses do not match the empirical rules (i) and (ii) above, the EPLR is zero.



(a) OWND vs Time



(b) Empirical rule (i)



(c) Empirical rule (ii)

Figure 7.6: L-Rex empirical rules and algorithm. OWND is the outstanding window, IRG is the inter-recv() gap, RRB is the number of recv()-returned bytes.

Example Describing How the Likely-Rexmit Model Works An example of how the L-Rex algorithm works is shown in Figure 7.6a, Figure 7.6b and Figure 7.6c. These three Figures are relative to a Cubic TCP connection with RTT equal to 259 ms and (nominal) bottleneck bandwidth equal to 7 Mb/s (ADSL). The sender was a Linux 3.10.17 box attached to our Campus LAN, the RTT was artificially inflated using Netem, and the receiver was a MacOS 10.8.5 MacMini attached to the 7 Mb/s ADSL via a 11 Mb/s WiFi network. The OWND plot was created by running Tcptrace on a packet trace captured at the sender, while the IRG and the RRB plots were created from application-level data saved on the receiver side by TestbedKit. Figure 7.6b reveals a few larger-than-threshold IRGs and Figure 7.6c has at least two RRBs that are clearly non-typical. The intersection of the larger-than-threshold IRGs and of the non-typical RRBs yields two likely-retransmission

points that can be easily mapped onto the two fast-retransmit CWRs[9] shown in Figure 7.6a.

We conclude this example with an observation useful to understand why L-Rex needs two empirical rules. Figure 7.6b shows that at the beginning of the transfer there are a number of samples for which the IRG is higher than the $H \cdot RTT$ threshold (which, for $H = 0.7$ and $RTT = 0.259s$, is equal to $0.183s$). The reason why there are those high IRGs is that at the outset TCP attempts to estimate the maximum number of outstanding packets that the network can hold, by quickly increasing the number of outstanding packets with the slow start algorithm. In particular, in the first burst of packets TCP sends two packets because the initial value of the CWND is two. After those two packets are sent, the sender must wait one RTT (by definition of RTT) to receive the first ACK, after which it can send more packets: as a consequence, the receiver sees a one-RTT gap between the first and the second burst of packets. Because the sender sends increasingly more packets in each burst, the fraction of the RTT in which the receiver is idle becomes smaller and smaller, until the pipeline is full and the receiver sees a continuum of equally spaced packets. Because the high IRGs are not only a characteristic of a retransmission but also of the initial slow start, rule (i) is not enough to detect likely retransmissions. However, to the rescue comes rule (ii), which allows us to identify retransmissions, knowing that a retransmission is often followed by a non-typical number of bytes.

Differences (and Affinities) between Inv-M and L-Rex The L-Rex model is more complex than the Inv-M model, because it gathers data each time that recv() returns and because, to compute the EPLR, it needs to process such data at the end of the transfer. However, the L-Rex model should be better suited than Inv-M to estimate the PLR of connections in which there is high bufferbloat delay. Moreover, unlike Inv-M, L-Rex should be able to detect cases in which there are no losses; furthermore, L-Rex accuracy should not depend on the congestion avoidance algorithm used: in fact, most (if not all) TCP flavors perform a fast recovery that takes, at a minimum, one RTT[10]. However, L-Rex may not be equally accurate for all access networks, because we specifically optimized it for ADSL and Fast Ethernet access networks under moderate network losses. Anyway, the moderate losses requirement is also a requirement of Inv-M, therefore neither models are suitable to estimate the PLR of a TCP connection that, due to the high packet losses, is working in stop-and-wait mode (i.e., never really exits the slow start). Likewise, both models assume that the sender is sending at the maximum possible speed, therefore they are both applicable to, e.g., FTP downloads and single-connection HTTP downloads; at the same time, they should not be applicable to protocols that exchange many small messages (e.g., BitTorrent).

[9] Congestion Window Reductions, defined in Section 7.2.4.

[10] As part of our future work, we should investigate whether the recovery time is shorter when explicit congestion notification (ECN) [267] – which however is not widely used – is enabled.

7.6 Comparison of Likely Rexmit and Inverse Mathis

In this Section we compare the accuracy of the L-Rex model with the accuracy of the Inv-M model. We will proceed as follows: we start by describing the 'raw test', a Neubot test that we used to collect data useful to compare the two models; then, we describe the results of controlled Internet experiments; afterwards, we compare the two models using testbed experiments; finally, we explore the results of the large-scale Internet experiments performed by the Neubot instances installed worldwide.

The Raw Test To produce data useful to compare the Likely Rexmit (L-Rex) model to the previously-developed Inverse Mathis (Inv-M) model, we wrote and deployed a new ad-hoc Neubot test, called 'raw test'. This test is an active transmission test that connects to a random M-Lab server (on port 12345) and performs a 10-second random-data TCP download. The main difference between the 'raw test' and the TestbedKit is that the TestbedKit was a standalone piece of software written for running simple TCP experiments, while the 'raw test' is a Neubot module, written using the low-level Neubot APIs and following the typical negotiate-test-collect Neubot test flow.

During the 'raw test', the sender (which is expected to run on an M-Lab [206] server in the context of the Neubot server process) saves TCP state variables, exported by Web100 [251]. Notably, it saves: (a) *CongestionSignals*, the number of congestion signals experienced by TCP (which, consistently with our definition of congestion signals, includes both timeouts and early congestion signals); (b) *SegsOut*, the total number of segments sent by TCP, including retransmitted ones. Meanwhile, the receiver saves the following application-level data: (c) the *connect delay*; (d) the *goodput*; (e) the *maximum segment size*, obtained via getsockopt(); (f) a list that contains the number of bytes returned by each recv() and the time elapsed since the previous recv().

The (a)-(f) data above allows us to compute both the Inv-M and the L-Rex PLR, using the procedures described in Section 7.3.2 and 7.5. It also allows us to compute the real PLR (RPLR) by dividing *CongestionSignals* by *SegsOut*: this is the PLR experienced by the transmitter, and we will use it as the ground truth to evaluate both models.

7.6.1 Controlled Internet Experiments

In this Section, we discuss the results of 'raw test' experiments performed on access networks for which we could capture packets (called 'controlled experiments' because we had full control over the computer where Neubot was installed). The objective is to investigate how both models approximate the Real PLR (RPLR).

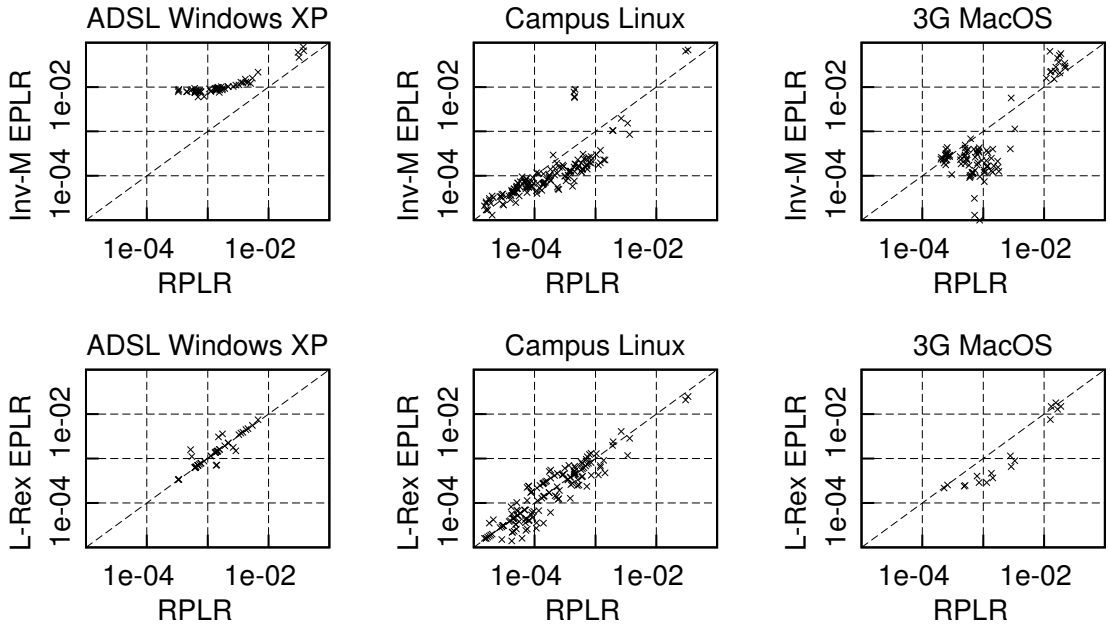


Figure 7.7: Estimated PLR (EPLR) versus the real PLR (RPLR).

Qualitative Overview Nine controlled experiments on a combination of Internet connections (ADSL, Fast Ethernet and 3G) and platforms (Linux, Windows, and MacOS)[11] were run. Neubot testing policy was modified to start a new 'raw test' every 15 seconds. Tests were performed towards random M-Lab servers. Each experiment is relative to one connection and one operating system, and contains at least 100 tests.

Fig. 7.7 shows the results of three experiments for both the Inverse Mathis (Inv-M) and the Likely Rexmit (L-Rex) models. The dashed bisector line represents the exact prediction by a model that covers the whole RPLR range. Therefore, the closer to the bisector the points are, the more accurate the model is.

Inv-M points are reasonably close to the bisector, but do not evenly distribute around it. L-Rex points, instead, are considerably closer to the bisector and are more evenly distributed. The points distribution suggests that we can reduce the error by averaging multiple experiments on the same network path.

Windows XP ADSL points differ significantly between the two models, with L-Rex points much closer to the real PLR. Tcptrace reveals that Windows XP performance is in most cases (79%) limited by the receiver's buffer; therefore, the Mathis model assumptions are not met, and the corresponding EPLR cannot be trusted.

[11]M-Lab servers used NewReno TCP throughout this experiment campaign. Windows 7 and Linux automatically scaled their TCP buffers, while MacOS 10.6 and Windows XP had fixed buffers.

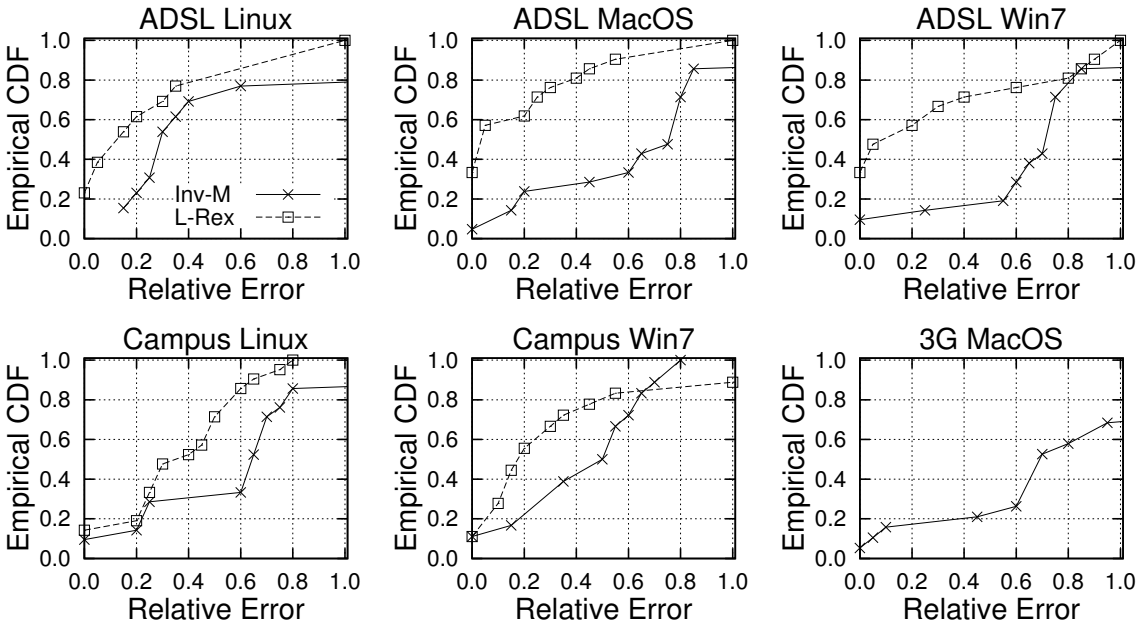


Figure 7.8: Controlled Internet experiments: empirical cumulative distribution function (CDF) of the relative error for diverse operating systems and access networks.

Median Behavior We showed that the L-Rex model is more accurate than Inv-M, with points evenly distributed around the bisector. Now, the question is whether we can reduce the noise and obtain a robust estimate of the typical PLR of a network path. As an indication of that, we use the median PLR.

We decided to use the median, and not the average, because the former is particularly robust to outliers. The PLR, in fact, can potentially range over a few orders of magnitude, and we want to avoid that occasional large values skew the estimate; e.g., if the typical PLR is $5 \cdot 10^{-4}$, one single sample equal to 10^{-3} can considerably shift the average.

To study the models median behavior, we performed repeated controlled 'raw test' experiments with seven M-Lab sites (corresponding to 21 servers). They were selected to represent near (i.e., same continent) and far (i.e., transoceanic) destinations. From the results we computed the RPLR, the Inv-M EPLR and the L-Rex EPLR.

Successively, we grouped experiments performed by the same client with the same server together, and we computed the medians (ignoring the cases with less than four samples).

Comparison of the Models Fig. 7.8 shows the empirical cumulative distribution function (CDF) of the relative error, computed on the medians (there are at least 13 points for each CDF). The distribution shows that L-Rex is more accurate than Inv-M. There is a good probability that the L-Rex error is limited; e.g., in the ADSL Linux case, for more than 75% of the samples the relative error is lower than 0.4. Interestingly, in Fig. 7.8, no CDF is plotted for 3G MacOS L-Rex; we investigated and we noticed that the relative error is 100% (i.e., the median EPLR is zero) in 18 cases out of 19. This happens because

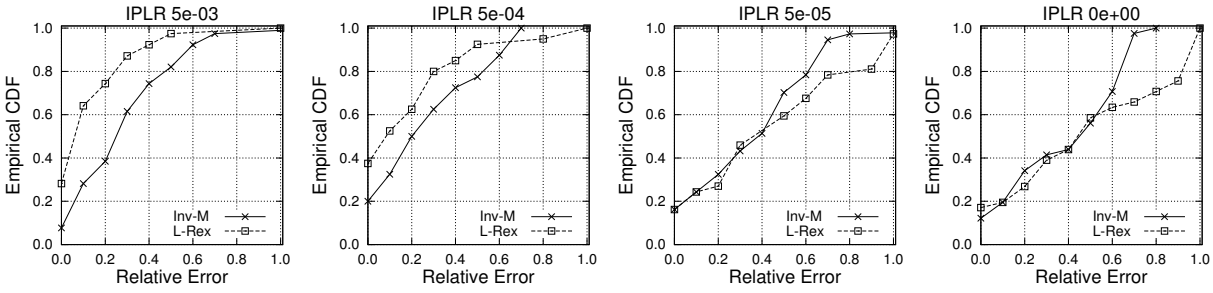


Figure 7.9: Testbed experiments: empirical cumulative distribution function (CDF) of the relative error for varying IPLRs (Netem-imposed PLRs).

the variance of the number of bytes returned by recv() was higher than it is for the ADSL and Fast Ethernet connections for which we optimized L-Rex; therefore, the 1% threshold was too strict to detect the vast majority of the losses. In our future work, we look forward to replace the static threshold on the number of bytes returned by each recv() with a more dynamic rule that depends on the distribution of the number of received bytes.

We also investigated the effect of the bufferbloat delay on the Inv-M accuracy. To do that, as anticipated in Section 7.3.3, in the Inv-M formula we replaced the connect delay with the average Web100 RTT, thus computing the B-IM (for 'Benchmark Inv-M') EPLR. This change reduced the error, but the model was still not as accurate as L-Rex, suggesting that the bufferbloat delay is not the only reason why the Inv-M model is less accurate than the L-Rex model (we will return to this point in Section 7.6.3).

7.6.2 Testbed Experiments

In the previous Section we tested the Inv-M and the L-Rex models on access networks that we controlled. However, we did not control the typical loss rate of the end-to-end path, which was a mixture of bursty losses (caused by the gateway queue dynamics) and much-less-bursty background losses, typical of the traversed (long distance) networks. In this Section, instead, we use Netem to add random losses to the typical background losses of a network path. This allows us to study how the model behaves when the background loss rates of a network path increases (simulating what happens when there is congestion or when an ISP is actively managing router queues).

To run the experiment, Neubot was installed on a Linux box which was attached to a Linux gateway in our campus Fast Ethernet network. The gateway was configured to discard packets with a random loss model and no correlation, using Netem. Successively, we run 'raw test' experiments with varying IPLRs. We varied the IPLR from 0 (i.e., the PLR was the typical PLR of each network path) to $5 \cdot 10^{-3}$ (where the IPLR dominated over the typical PLR of every path we tested).

Fig. 7.9 shows the empirical cumulative distribution function (CDF) of the relative error. Each experiment consists of at least 37 samples, with only three experiments with RPLR equal to zero (two for IPLR equal to $5 \cdot 10^{-5}$ and one for IPLR equal to zero). The

trend is that L-Rex is more accurate than Inv-M when the IPLR is higher than $5 \cdot 10^{-5}$; for lower IPLRs, L-Rex is less accurate than Inv-M. This trend reflects the general behavior of the models for different RPLR ranges, as we will see in the next Section.

7.6.3 Large-Scale Experiments

In this Section we analyze the results of 'raw test' experiments run from the access networks of the 1480 users that have installed Neubot. The objective is to study the models accuracy at different RPLR ranges (including when the RPLR is zero).

Data Overview and Preprocessing Our dataset contains the results of the 'raw test' performed by Neubot instances installed worldwide. The dataset is in the public domain and can be downloaded freely from the Neubot web site [103]. From 18th November 2012 to 17th February 2013, 565,559 tests were performed by 22,681 unique IP addresses and by 1,480 unique Neubot IDs, using four diverse platforms: Win32 (259,910; 46% of the tests), MacOS (228,712; 40%), Linux (75,497; 14%), and FreeBSD (1,440; 0.25%). We will concentrate on the first three cases. Tests were performed towards 33 M-Lab sites (corresponding to 99 distinct servers), with a nearly uniform distribution of the load among them. Countries with more tests are: the U.S. (195,059; 34%), Italy (92,210; 17%), Germany (67,496 11%), France (32,098; 5%), and the U.K. (27,271; 4%).

Neubot does not gather information on the type of access network used by the device on which it is installed (i.e., whether it is fixed or mobile). However, a quick analysis of the involved Autonomous Systems shows very few mobile providers. From the above data, we computed the RPLR, the Inv-M and the L-Rex EPLRs.

As anticipated in Section 7.3.3, we also estimated the PLR with Inv-M using the average Web100 RTT in place of the connect delay. We will label this case as the 'Benchmark Inverse Mathis' (B-IM). As in Section 7.6.1, we compute the medians to reduce the noise caused by outliers. We group together experiments performed by the same client with an M-Lab site, and we compute the median EPLR and the median RPLR (ignoring the cases with less than eight samples).

Relative Error Fig. 7.10 shows the empirical cumulative distribution function (CDF) of the relative error computed on the medians, for different RPLR ranges and operating system platforms. Unlike Section 7.6.1, here we can not distinguish between Windows 7 and XP because Neubot identifies only the platform. Tab. 7.4 shows the number of points used for the CDF.

In Fig. 7.10, for high RPLR (i.e., greater than 10^{-3}) and moderate RPLR (i.e., between 10^{-4} and 10^{-3}), the L-Rex model error is reasonably limited, and the model is more accurate than Inv-M: in the worst case (Win32 for moderate RPLR), in fact, L-Rex has lower relative error than Inv-M for 65% of the samples. However, for low RPLR (i.e., lower than 10^{-4}), it is less accurate than the Inv-M model. As we noted in Section 7.5, in fact, the L-Rex model was designed and optimized for landline networks with moderate losses. This explains why

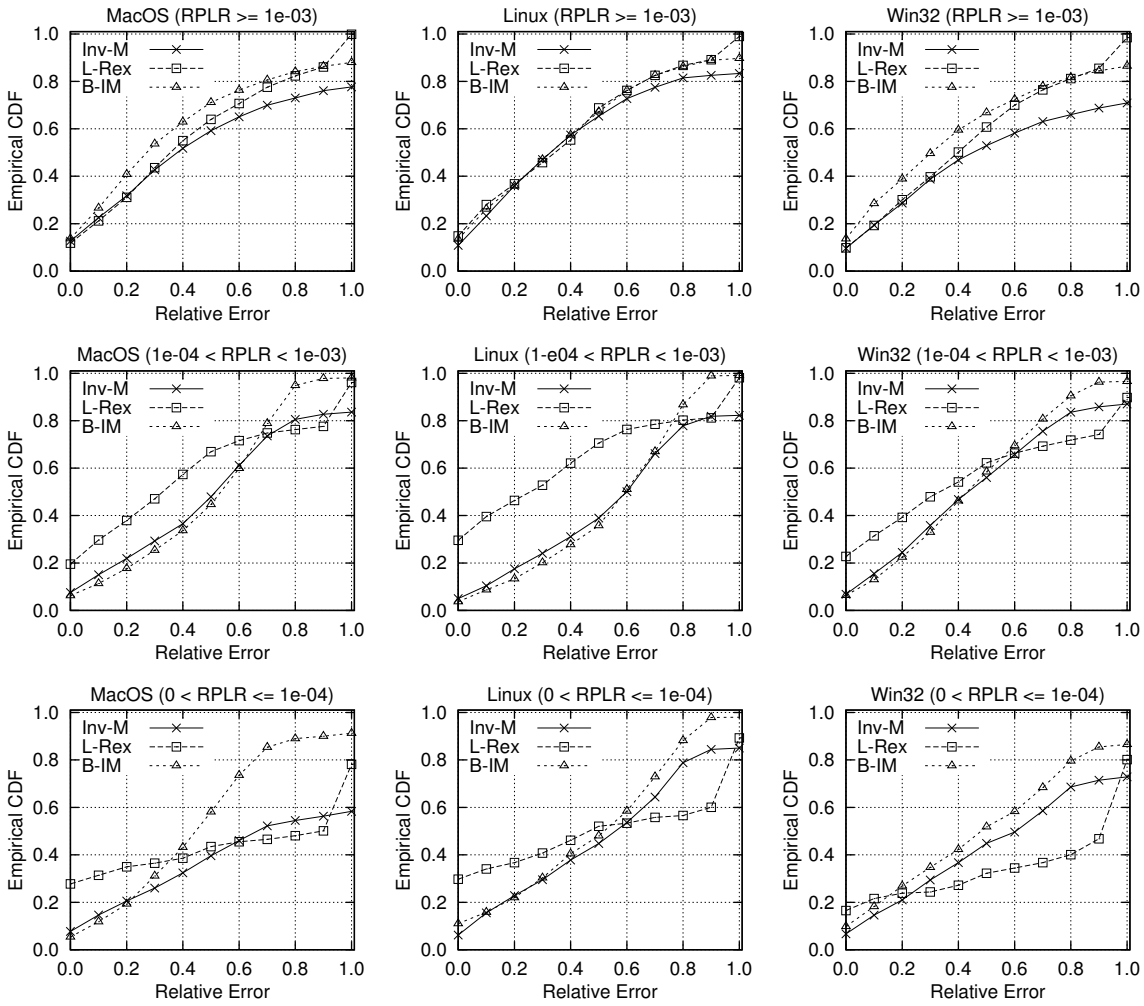


Figure 7.10: Empirical cumulative distribution function (CDF) of the relative error for different RPLR ranges and operating system platforms.

it is more effective than Inv-M for RPLR greater than 10^{-4}. The large-scale experiments confirm that the bufferbloat delay reduces the Inv-M accuracy. Indeed, when we use the Web100 average RTT in place of the connect delay, we typically obtain a more accurate estimate (indicated as B-IM in Fig. 7.10). In particular, for Win32 and MacOS, B-IM is the most accurate model for both high and low RPLR conditions.

We performed one extra experiment on a Gigabit Ethernet with default MSS (i.e., 1,460 bytes) to understand L-Rex reduced accuracy with low RPLR. We noticed that the variance of the number of bytes returned by each recv() was higher than it is for the ADSL and Fast Ethernet connections for which we optimized L-Rex; therefore, the 1% threshold was too strict to detect all the losses that occurred.

RPLR	MacOS	Linux	Win32
$[10^{-3}, \infty)$	1,456 (24%)	378 (20%)	1,529 (22%)
$(10^{-4}, 10^{-3})$	2,179 (35%)	839 (44%)	2,115 (30%)
$(0, 10^{-4}]$	389 (6%)	373 (19%)	357 (5%)
0	2,148 (35%)	333 (17%)	3,039 (43%)

Table 7.4: Number of points in Fig. 7.10 (first three rows) and Fig. 7.11 (last row).

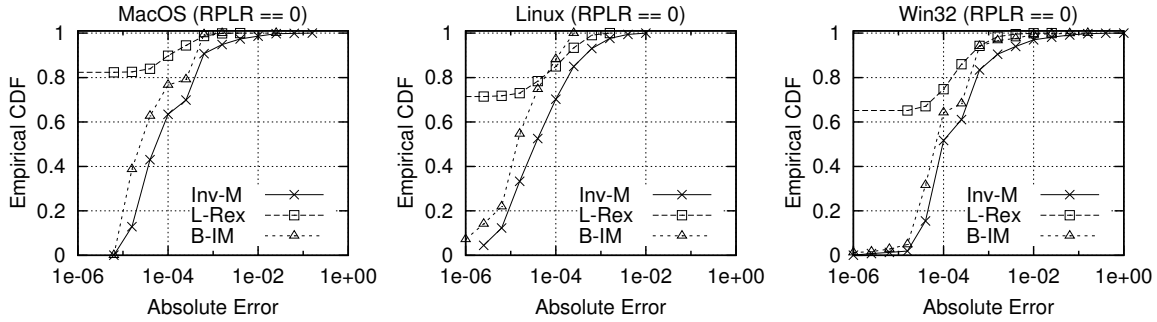


Figure 7.11: Large-scale experiments: Empirical cumulative distribution function (CDF) of the absolute error for different operating system platforms when RPLR is equal to zero.

Absolute Error when RPLR is Zero To complete our comparison of the two models, we study now the case when the RPLR is zero. We did not study it before because in controlled and testbed experiments the case was not frequent. Instead, as Tab. 7.4 shows, this case is more significant in large scale experiments. Fig. 7.11 shows the empirical cumulative distribution function (CDF) of the absolute error for different operating system platforms when the RPLR is equal to zero. The error is computed on the medians and is equal to the EPLR (since the RPLR is zero). Tab. 7.4 shows the number of points used to prepare the cumulative distribution function.

When the RPLR is zero, L-Rex is the best model, because in the worst case (Win32) it yields zero 65% of the times. On the contrary, Inv-M is much less accurate, but this should not come as a surprise: as we anticipated in Section 7.3.2, in fact, the model EPLR cannot be zero. The distribution also shows that the L-Rex model is less likely to yield high estimates when the RPLR is zero. In general, L-Rex is the most accurate model (equally with B-IM for Linux). In the worst case (Win32), for 25% of the samples the model yields a value greater than 10^{-4} when the RPLR is zero.

Comparison of the Models Large scale experiments results confirm that L-Rex is more accurate than Inv-M to estimate moderate to high PLR. However, L-Rex is less accurate than Inv-M for low PLR. L-Rex is also good at detecting when there are no losses (a case that Inv-M is not able to detect). Inv-M, instead, is consistent along the whole PLR range but its precision is reduced by the bufferbloat delay. This suggests that the model can be improved by using a better average RTT estimation, yet, since B-IM (which uses Web100 average RTT) is still less accurate than L-Rex for moderate RPLR, the bufferbloat delay

seems not to be the only reason why Inv-M is less accurate than L-Rex.

7.7 Concluding Remarks

In this Chapter we described two models to estimate the packet loss rate (PLR) experienced by a TCP connection from application level measurements: the Inverse-Mathis (Inv-M) model and the Likely-Rexmit (L-Rex) model. The two models are suitable to estimate the PLR experienced by a continuous steady-state TCP flow (e.g., a download performed using FTP as well as a download using a single HTTP connection), but are not suitable to model the performance of protocols that send many small messages (e.g., BitTorrent).

We started the chapter by explaining why it is interesting to study models for estimating the PLR experienced by a TCP connection from application-level measurements. To account for our interest in TCP, we explained that the opaque TCP/IP stack is increasingly becoming a limit both for developers and researchers studying network neutrality, because it hides relevant information (e.g., losses and RTT changes) to which, instead, the network has easy access. As regards the interest on application-level measurements, they are very appealing because they can potentially reach many more users than, say, measurements that require to install a modified kernel. Moreover, even though they are intrinsically less accurate, application-level measurements could be just one of the input signals of a multi-layered measurement system, a signal perhaps used to trigger more accurate experiments when anomalies arise.

After the explanation of our motivations we provided a brief review of TCP to encompass the most important concepts needed to understand the remainder of the Chapter. We covered, in particular: the basics (packets, ACKs and SACKs); a brief history of TCP from the seminal RFC793 to NewReno TCP; Cubic TCP, an alternative high-speed flavor of TCP that is the default TCP on Linux and that is used in some of our experiments; a few more definitions, including the definition of bufferbloat and bufferbloat delay (in short, the bufferbloat delay is the very high queuing delay that a TCP connection creates on a slow gateway, which in turn affects the RTT of the connection).

Next, we studied the Inv-M and the L-Rex models, with diverse experiments that range from testbed experiments to controlled and large-scale Internet experiments. We dedicated a Section to the analysis of the Inv-M model performance, at the end of which we listed its main problems: the small (residual) dependence of the PLR estimate on the RTT when TCP Cubic is used; the fact that the simple Inv-M formula does not consider the case in which there are no losses; the adverse effect of the bufferbloat delay on the PLR estimate. Afterwards we dedicated a Section to the L-Rex model, which we designed to overcome some of the limitations of Inv-M, and we performed a comparison of the two models using a Neubot experiment called 'raw test' and M-Lab servers. Overall, we found that the L-Rex model (more complex than Inv-M because it computes the EPLR estimate from data collected after each recv()) is more accurate than Inv-M for PLRs above 10^{-4} and for the ADSL-and-Fast-Ethernet connections for which it was optimized. Inv-M, on the contrary, is more consistent across the whole PLR range; still, it cannot detect the absence of losses,

a task in which L-Rex is instead quite accurate. In conclusion L-Rex is an improvement over Inv-M and, albeit not equally accurate in all scenarios, it should allow one to correctly estimate the magnitude of the PLR in a large number of real cases.

Regarding future work, we noted a few access networks in which L-Rex accuracy is slightly lower (e.g., Gigabit Ethernet) or significantly lower (e.g., mobile) than expected. It turns out that, in such access networks, the static L-Rex thresholds (which were optimized for ADSL and Fast Ethernet) are not adequate. To solve this problem, we plan to replace the static thresholds with dynamic rules that depend on the characteristics of the access networks. At the same time, we also plan to perform controlled Internet experiments in cable networks (which are, with ADSL and Fast Ethernet, the most commonly used access networks). Finally, we believe that it should be possible to improve the accuracy of L-Rex by taking into account the temporal correlation of the data collected after each recv().

Chapter 8

Experiment #3: Dynamic Adaptive Streaming over HTTP

This Chapter describes a Neubot test, called 'Dashtest', that emulates the Dynamic Adaptive Streaming over HTTP (DASH) standard [227] to collect a dataset useful to evaluate the performance of diverse rate-adaptation algorithms for DASH streaming. There is interest in the DASH standard because the DASH is currently used by popular video streaming services (e.g., YouTube, Netflix, and Hulu) to stream video content. In addition to being useful to the research community that studies DASH-based multimedia streaming, the Dashtest is also very relevant to our main research thread: in fact, it allows us to benchmark the users' access networks with streaming, which accounts for the majority of consumer Internet traffic. Moreover, the Dashtest also runs for longer time than other Neubot tests; therefore, it provides interesting insights into the sustained performance that an access network can provide. Finally, the Dashtest deployment confirms the fitness of the Neubot architecture for the purpose of running experiments from the edges: we developed and deployed, in fact, the Dashtest test with limited effort and we were able to collect a large amount of data in a short time-frame.

Based on the paper presented at the 2014 Multimedia Systems (MMSys) conference [179], this Chapter is organized as follows: Section 8.1 describes what motivated us, the related work and the intended usage of the dataset collected using the Dashtest (henceforth, 'the dataset'); Section 8.2 describes how we implemented the Dashtest, how the Dashtest is integrated into Neubot, and how much time it required to develop and deploy the Dashtest; Section 8.3 focuses on the dataset, providing statistics, describing the data format and showing how the dataset can be used to simulate diverse rate-adaptation algorithms; finally, Section 8.4 draws the conclusion and describes possible future developments.

8.1 Motivation

Recent years have witnessed a strong shift towards media streaming based on the HTTP protocol. This trend significantly changed the traditional paradigm in which generic web

content (hypertexts, pictures, etc.) and media (particularly audio and video) were delivered using two different strategies. Historically, in fact, web content was transferred using the HTTP protocol and huge investments were made to ensure fast delivery of popular content to the end users, relying on caching proxies and content delivery networks (CDNs). Conversely, media were mostly transferred using UDP-based protocols specifically designed for streaming, e.g., the Real Time Protocol (RTP) [280]. The fact that media streaming was not transferred using HTTP, however, prevented it from using the highly-optimized architecture already used for web content, in which many companies had been heavily investing.

Therefore, since 2007 the HTTP protocol is being increasingly used to deliver media, using a strategy that was originally called 'adaptive HTTP streaming'. The key idea of this strategy is the following: to adapt a video (or audio) stream to the resource-oriented web paradigm, a media file is divided in small chunks (often called 'segments'), each addressable by its own URL. Moreover, each chunk is available under diverse bitrate representations, allowing the client to choose the representation that optimizes the user's quality of experience (QoE). Since the HTTP protocol was used, this strategy allowed (and still allows) the content providers and the ISPs to benefit from the investments already made to optimize web-content delivery. At the same time, however, the adaptive HTTP streaming shifted the burden to manage the media transfer to the client: to maximize the QoE, in fact, the client not only has to constantly monitor the download speed but also has to select the most appropriate representation of each video segment (a task that with RTP/RTSP was, conversely, responsibility of the streaming server).

At the outset many adaptive-HTTP-streaming solutions were independently developed by major companies (Microsoft with Smooth Streaming in 2008, Apple with HTTP Live Streaming in 2009, Adobe with HTTP Dynamic Streaming in 2010). Although the basic idea was practically the same, however, these solutions were incompatible among them due to lack of standardization. To overcome this problem, in April 2012 the MPEG Working Group ratified a new standard called 'Dynamic Adaptive Streaming over HTTP' (or, simply, DASH) [227]. Following the MPEG standard philosophy, the DASH enables interoperability and fosters adoption by the industry, yet it still allows for competition among different companies, which are free to optimize various implementation aspects. In particular, the DASH standard does not specify the rate-adaptation algorithm, which is key to guarantee the QoE, but which has no impact on interoperability.

To date most of the open-source implementations that support the DASH are based on the Libdash library [74], which provides only two basic rate-adaptation algorithms: manual adaptation and always-lowest bitrate. Other open-source implementations such as the ones bundled with the Mozilla Project products [92] and with the open-source VLC project [142], instead, rely on the session-average-bitrate (SAB) algorithm in which the software selects the next-chunk representation closer to the average bitrate of the whole streaming session.

While open-source implementations use simple algorithms, the literature contains many examples of papers that propose and compare advanced rate-adaptation algorithms. For

instance Liu, et al., [241] propose to use the ratio of the segment duration and the latest single-segment fetch time to detect network congestion and spare network capacity; this technique is evaluated using the network simulator ns-2 in emulated networks with background traffic generated by constant-bitrate exponentially-distributed sources. Other papers, instead, focus on real-networks experiments: Akhshabi, et al., [170] experimentally evaluate the rate adaptation algorithms of three major video players (Smooth Streaming, Netflix, OSMF); Muller, et al., [257] compare their implementation to other proprietary systems in a vehicular-network scenario.

HTTP-based adaptive streaming solutions are increasingly chosen by several companies as the delivery method for streaming media: a first live-and-large-scale DASH deployment occurred during the 2012 London Olympics, with up to 1,000 concurrent viewers; successively, Wimbledon and Roland Garros followed suit. Popular video streaming services (e.g., YouTube, Netflix and Hulu) are also, reportedly, DASH based[1].

However, neither the above-mentioned studies nor the above-mentioned commercial implementations publicly released data regarding the network conditions experienced by DASH; similarly, no information on the rate-adaptation logic is available; likewise, there is no dataset encompassing the measured download rates. As a consequence, it is extremely difficult to evaluate and/or compare different DASH algorithms, since a common dataset for the purpose is missing. In fact, we are aware of only two publicly-available DASH datasets, described in the works of Lederer, et al., [238] [239]. The former mainly consists of A/V material for video quality evaluation; in particular, it is composed of several videos with a duration between 10 and 90 minutes, in which each video is provided in diverse segment lengths as well as in diverse representations, ranging from 50 kb/s up to several Mbit/s. The latter is related to a distributed set of videos that is mirrored at many diverse sites across Europe (e.g., Klagenfurt, Paris, Prague), which allows clients to download video segments from those sites and dynamically switch to other sites, should they choose to do so. In particular, the dataset is suitable for testing the DASH in real-world scenarios in which, notably, a client can switch from one content-delivery network to another.

Consisting only of video material that can be streamed, the two datasets described above provide excellent facilities for running large-scale DASH experiments: to evaluate a novel rate adaptation algorithm using one of the two datasets, one needs to implement such algorithm for a given client (e.g., VLC [142]), to install such client on a number of computers (possibly on many computers around the globe using, e.g., using PlanetLab [195]), and to

[1]See, e.g., this November 18, 2013 Akamai blog post entitled "MPEG-DASH is now industry essential" [303], which reads: "YouTube is not the only actor to deploy DASH at large scale. Netflix has been promoting DASH for a long time now, and uses it in production on Chromium and Internet Explorer 11 versions of the service, with predictable extension of the compatibility scope as Netflix is pursuing a determined quest towards format and player de-duplication. Hulu's plans with DASH have also started to surface, with more details promised during the Streaming Media West show in November." Speaking of YouTube, it is certainly also worth mentioning YouTube's "MPEG-DASH / Media Source demo", which implements a demo DASH player, "provided for testing and validation purposes, but [...] not intended to serve as a public or standardized implementation or reference" [42].

instruct such computers to fetch the videos of the selected dataset. However, because they do not include network-level traces, the two datasets are not suitable to simulate the behavior of novel rate adaptation algorithms.

Motivated by the desire to foster the research on novel rate-adaptation algorithms, we contribute by publicly releasing a dataset containing network measurements collected using the new Neubot 'Dashtest'. Because the measurements involve random M-Lab servers and diverse access networks around the world, the dataset provides large-scale DASH-performance insights useful to simulate and compare diverse rate-adaptation algorithms. Moreover, our DASH dataset is also relevant from the network-transparency point-of-view, because it allows us to benchmark the users' access networks with the streaming, which accounts for the majority of consumer Internet traffic [32].

8.2 Methodology: the Neubot Dashtest Test

In this Section we describe how we implemented the Dashtest, how such test is integrated into Neubot, and how much time it required to develop and deploy the Dashtest. We start by describing the implementation of the new Dashtest.

Mod_dash: the Neubot DASH Module The Dashtest is implemented by mod_dash, a Neubot module that emulates the DASH streaming of a video resource composed of fifteen two-second segments. Each segment is available in one of the bitrate representations used by the dataset described in the work of Lederer, et al., [238] (100, 150, 200, 250, 300, 400, 500, 700, 900, 1200, 1500, 2000, 2500, 3000, 4000, 5000, 6000, 7000, 10000, 20000 kb/s), to ease the comparison between datasets.

Unlike other DASH implementations, there is no media presentation description (MPD) in Dashtest, rather the test server sends the list of available bitrates to the Neubot instance (henceforth, 'the instance') just before the beginning of a Dashtest. Also, unlike real videos, the video payload is emulated by a sequence of random bytes.

The mod_dash module is composed of a mod_dash test server (which registers a HTTP-request handler within the Neubot server) and a mod_dash client (henceforth, client, which runs in the context of the Neubot instance). The client connects to a random M-Lab test server and requests the fifteen two-seconds emulated-video segments using the rate-adaptation algorithm that we describe in the following paragraph.

The Dashtest Rate-Adaptation Algorithm The Dashtest implements a modified last-segment-bitrate (LSB) rate-adaptation algorithm: LSB is the simplest rate adaptation algorithm, because the rate is adapted taking into account the bitrate of the last-downloaded media segment only[2]; the modification is that the Dashtest slows down when

[2]The name LSB for this basic algorithm was not invented by us: this name was used, e.g., by Thang, et al., [294] in their 2012 paper for the International Conference on Communications and Electronics.

it estimates that the network conditions have worsened. Before we elaborate on the modification, however, let us give more details on the LSB algorithm in general and on its Dashtest implementation in particular.

The LSB algorithm works as follows. At the beginning the client requests the first segment using the most-conservative bitrate representation (also called MCBRR and equal to 100 kb/s in our case). Subsequent segments are requested using possibly-varying representations: after the download of each segment the client computes the estimated available bandwidth (EAB) dividing the size of the segment in bits by the download time in seconds; then, the client selects ρ, the highest-available representation bitrate (HRB) smaller than the EAB, and requests to the server the next segment encoded at ρ bit/s.

As regards the way in which Dashtest implements the LSB, the main difference between the theory and the implementation is the following: once the representation bitrate is known (be it the MCBRR or the HRB), Neubot multiplies it by two seconds (which is both the playout time of an emulated segment and the expected download time) to obtain β, the number of bytes to request to the server; then, the Dashtest client requests to the server a β-bytes-long segment. Of course, all the segments are requested using the same persistent HTTP connection, moreover, the interval between two consecutive requests is in general smaller than the TCP retransmit timeout (RTO); therefore, TCP should not exit the congestion avoidance state (unless network losses force TCP out of such state).

In addition, as anticipated above, there is also the modification with respect to the baseline LSB algorithm: a mechanism designed to reduce network usage when the Dashtest estimates that the network conditions have worsened. This mechanism is important to avoid monopolizing the Neubot user's access network during the Dashtest; in fact, because the Dashtest is a Neubot test, it runs in the background and shall avoid being too aggressive. For this reason this mechanism is designed such that, if the user opens one-or-more bandwidth-hungry foreground flows, the Dashtest will detect that the network conditions worsened and will attempt to reduce the network usage, thereby releasing network resources that the foreground flows can use. Of course, after the download of the next segment, the Dashtest will check again whether the network conditions worsened and will further reduce the network usage, if needed.

More in detail, the Dashtest assumes that the network conditions worsened when the download time of a segment is higher than the expected download time (EDT, equal to two seconds). In such case, the Dashtest reduces the EAB proportionally to the difference between the download time and the EDT, using the following algorithm, inspired to the linear controller used by the Low-Extra-Delay Background Transport (LEDBAT) [283]:

```
if EDT > PLAY_TIME:
  REL_ERR = 1 - EDT / PLAY_TIME
  EAB = EAB + REL_ERR * EAB
  EAB = max(MCBRR, EAB)
```

where REL_ERR is the relative error, EAB is the estimated available bandwidth, EDT is the download time of the just-downloaded segment, PLAY_TIME is the playout time of a segment (two seconds), and MCBRR is the most conservative bitrate representation (i.e.,

the minimum bitrate representation made available by the server, 100 kb/s).

Note that, because the linear controller reduces the EAB, the HRB should be reduced and the next-segment download speed should be reduced as well. In turn, downloading at reduced speed, we yield network resources to the user's flows. (To simplify the discussion, in the following we will also sometimes be slightly imprecise and say that "the download speed is reduced proportionally to REL_ERR." This statement is slightly imprecise because, on the one hand, the EAB adjustment may be so small that the HRB does not actually change and, on the other hand, the download speed is not actually reduced proportionally to REL_ERR, rather the reduction is only correlated with REL_ERR.)

By the way, the linear controller was not our first choice: to reduce the EAB, in fact, we initially considered a scheme in which the EAB was multiplied by a 0.5 factor. However, our experiments showed that such scheme was not adequate: it was too aggressive when the EDT was only marginally greater than the PLAY_TIME, and it was not aggressive enough when the EDT was significantly larger than the PLAY_TIME. On second thought, we implemented therefore the linear controller, which is more adequate to the task because the EAB is reduced proportionally to REL_ERR.

Let us now spend one paragraph to justify our choice of using a linear controller, to address the objection that, since we use a linear controller similar to LEDBAT, our rate adaption algorithm may well have the same well-known shortcomings that LEDBAT has. Recent research [193] demonstrated that LEDBAT has weak intra-protocol fairness properties because it reacts to excessive extra delay on the forward path (which for LEDBAT is a congestion signal) in a linear way rather than in an exponentially decreasing way. As a consequence, if there are two LEDBAT flows and one of them is using more than its fair share, the linear decrease does not slow down such flow enough to give the other flow a chance to increase its share of the bandwidth. However, the weak fairness is not a property of a linear controller per se, rather it is a property of a linear controller coupled with congestion avoidance, which is relevant to LEDBAT because LEDBAT aims to replace TCP for certain background tasks. Our work is significantly different from LEDBAT: we use a comparable linear controller, but unlike LEDBAT our linear controller works on top of TCP, therefore our Dashtest is as friendly as TCP is. In the same way, we can address similar objections that attribute to our rate-adaptation algorithm shortcomings that affect LEDBAT as a congestion avoidance protocol (e.g., that LEDBAT may measure its own induced delay when it periodically measures the base delay, so that the delay produced by a LEDBAT flow may continuously increase if the flow runs for long enough [271]).

To wrap up: the Dashtest selects the bitrate representation that guarantees a download time of about two seconds (which is the playout time of the emulated segment) and, when the download time is larger than two seconds, attempts to reduce the download speed proportionally to the difference between the download time and two seconds. In other words this rate-adaptation strategy seeks to minimize the playout buffer level, because it strives to keep the download time smaller than the playout time. However, as we will show in Section 8.3, the data saved by the Dashtest also allow one to simulate the performance of more complex rate-adaptation algorithms using the download speed of each chunk as an approximation of the instantaneous network conditions.

Name	last year	last 3 months
# of countries	146	108
# of Autonomous Systems (AS)	1,744	1,036
# of UUIDs	4,060	2,089
# of IP addresses	112,371	29,275
# of Different Locations	9,613	4,610
Median # of tests per IP	12	11
Median # of Dashtests per IP	7	7

Table 8.1: Dataset statistics as of November 11, 2013.

Complexity of Adding the Dashtest to Neubot In this paragraph we describe the complexity of adding the Dashtest to Neubot, to show that adding a new network experiment to Neubot is relatively simple, because Neubot is designed to host a variety of network experiments. In the Dashtest case, in particular, the work was particularly simple, because the implementation of the Dashtest was derived from the one of the 'HTTP speedtest' test (henceforth, speedtest, see Chapter 6). More specifically, we simplified the client-side speedtest implementation removing the code used to manage many concurrent flows. On the server side, we copied the speedtest server code and we easily adapted it with minor modifications. On the client side, we tweaked the code to save the results using the new pickle-based format rather than the old sqlite3 database (see Chapter 4), because, with respect to sqlite3, pickle was more scalable (in terms of number of records that we can save on disk without degrading the performance) and more easily tractable (allowing us to store the results into a circular buffer). We also copied and adapted the code to create a new Neubot module, including the JSON files to tell Neubot which plots to show in the web interface. Overall the new Dashtest is composed of 32,582 bytes of Python code and 4,161 bytes of HTML and JSON. It took us seven days to develop and test the new Dashtest (including the time dedicated to decide how to slow down when the network conditions worsened), plus other seven days to test the M-Lab deployment of the new test.

8.3 Description of the DASH Dataset

This Section describes the measurements collected by the Dashtest and discusses some insights that can be extracted from the measurements, laying the foundations for further studies concerning real-world DASH-streaming-performance measurements.

Data Structure and Data Collection Process In this paragraph we describe the data structure and we briefly recap how the data collection process works. Because the data collection process is not the main subject of this Chapter, however, we only provide here the bare minimum to understand how Dashtest data is collected. For a more extensive description, we refer the interested reader to Chapter 4.

Dashtests are performed with M-Lab servers chosen at random; in the following, we

will also call 'test server' the server with which a test is run. A Dashtest lasts about 30 seconds because it downloads fifteen segments and because the download of each segment should take about two seconds. The result of a Dashtest is a set of properties that describe the Neubot client and the test server, as well as the performance measured during the download of each emulated video segment.

Table 8.2 shows the results saved by a Dashtest. We can divide the results data into three sets: the first set, from *uuid* to *remote_address*, describes the Neubot client and the test server; the second set, from *whole_test_timestamp* to *connect_time*, describes the current Dashtest (to uniquely identify a test one can use the *srvr_data.timestamp* and the *remote_address* fields); the third set, from *iteration* to *delta_sys_time*, describes the download of each segment (also called 'iteration' of the test in Table 8.2).

As explained in Chapter 3, after a test the client and the test server merge the data they saved; the resulting JSON file is then stored both on the client and on the server sides. Once per day, the M-Lab data-collection pipeline fetches the JSON files from the test servers, aggregates them and publishes them on the Google-Cloud-Storage service under a No-Rights-Reserved Creative-Commons-Zero 1.0 Universal waiver [30].

More specifically: on the Google-Cloud Storage the JSON files are aggregated in tarballs that contain the JSON files collected during a day by a specific M-Lab tool (e.g., NDT, Glasnost, Neubot) running on a specific M-Lab server. For example, the tarball 20131008T000000Z-mlab1-lga01-neubot-0000.tgz contains the Neubot data saved on the mlab1-lga01 M-Lab server on October 8, 2013. The M-Lab wiki publishes a page explaining the many ways in which one can download the tarballs [85].

Overview of the Available Data As of November 2013 Neubot counts about 1,000 active users each day and a slightly-higher number of IP addresses: in fact, a number of users installed Neubot on their laptops, which are of course used from multiple locations (e.g., from home and from work). The number of tests run each day is about 20,000 comprising the four tests implemented by Neubot. Other statistics are presented in Table 8.1, and up-to-date statistics are available online [82]. The Dashtest, in particular, performs about 10,000 tests per day involving more than 1,000 IP addresses from about 100 countries and 1,000 autonomous systems (very few of which belong to mobile providers).

Figure 8.1 shows the location of the Neubot tests performed in October 2013; a more extended representation covering the 10,000+ different IP addresses and the 2,000+ diverse locations from which Neubot test have been performed is available online [83]. This kind of analysis is only possible because Neubot users provided us with the authorization to collect, publish and use their IP address for research purposes. From the IP address, in fact, we can easily determine the Autonomous System; moreover, using geolocation software such as MaxMind's GeoLite [11], we can also determine the timezone, the country, the region, the city, the latitude and the longitude. See Chapters 3 and 4 for a discussion of the privacy aspects of Neubot.

Because Neubot is an always-running background process that schedules two tests per hour, in many cases more than two Dashtests per day are performed. Figure 8.2 helps us

Name	Example	Description
uuid	7528d674-25f0-4ac4-aff6-46f446034d81	Neubot random unique identifier.
platform	linux2	The operating system platform.
version	0.004016008	Neubot version number.
real_address	130.192.225.141	Neubot IP address, as seen by the server.
internal_address	130.192.225.141	Neubot IP address, as seen by Neubot.
remote_address	80.239.142.212	Server IP address.
whole_test_timestamp	1382434858	Time when the test was performed (Unix epoch time).
srvr_data.timestamp	1382434858	Time when the test was performed on the server (Unix epoch time).
timestamp	1382434858	Time when the test was started (Unix epoch time).
clnt_schema_version	3	Version of the client schema.
connect_time	0.02469491958618164	RTT estimated by measuring the time that connect() takes to complete, measured in seconds.
iteration	14	Sequence number of the current download request.
request_ticks	1382434858.103292	Time when the request was served by the server (Unix epoch time).
elapsed_target	2	Expected download duration, measured in seconds.
rate	20000	Segment representation rate for the current request, measured in kb/s.
elapsed	0.5023031234741211	Time elapsed from the download request to the end of the download (i.e. download duration), measured in seconds.
received	5000131	Amount of bytes received from the server for the current request, measured in bytes.
delta_user_time	0.06000000000000005	Accumulated user time during a request, measured in seconds.
delta_sys_time	0.06000000000000005	Accumulated system time during a request, measured in seconds.

Table 8.2: Dashtest data format.

to study the distribution of the number of tests per IP address and per hour: the topmost plot shows the hour of the day in which Dashtests were performed as a function of the IP address index number (IP addresses are sorted by the number of tests performed in descending order); the middle plot shows the distribution of the number of tests per day as a function of the IP address index number; the bottom plot shows the distribution of the number of tests per hour. From the first two plots we can conclude that more than half of the IP addresses run more than seven tests per day (e.g., the IP #470 run Dashtests at 2:30, 5:11, 6:50, 8:39, 12:32, 16:31, 17:02). The bottom plot tells us that Dashtests are almost equally distributed during the day, with an average of 360 Dashtests per hour.

As a concluding remark, let us underline that the deployment of the Dashtest confirmed that the Neubot architecture is adequate for the purpose of measuring network-performance from the edges. Not only (as pointed in the previous Section) it was simple to develop and deploy the Dashtest but also we were able to collect a large number of datapoints in a very short time frame. Consider, in fact, than the new version of Neubot including the Dashtest was officially released on October 29, 2013 and on November 2, 2013 Neubot was already performing (on the average) 360 Dashtests per hour.

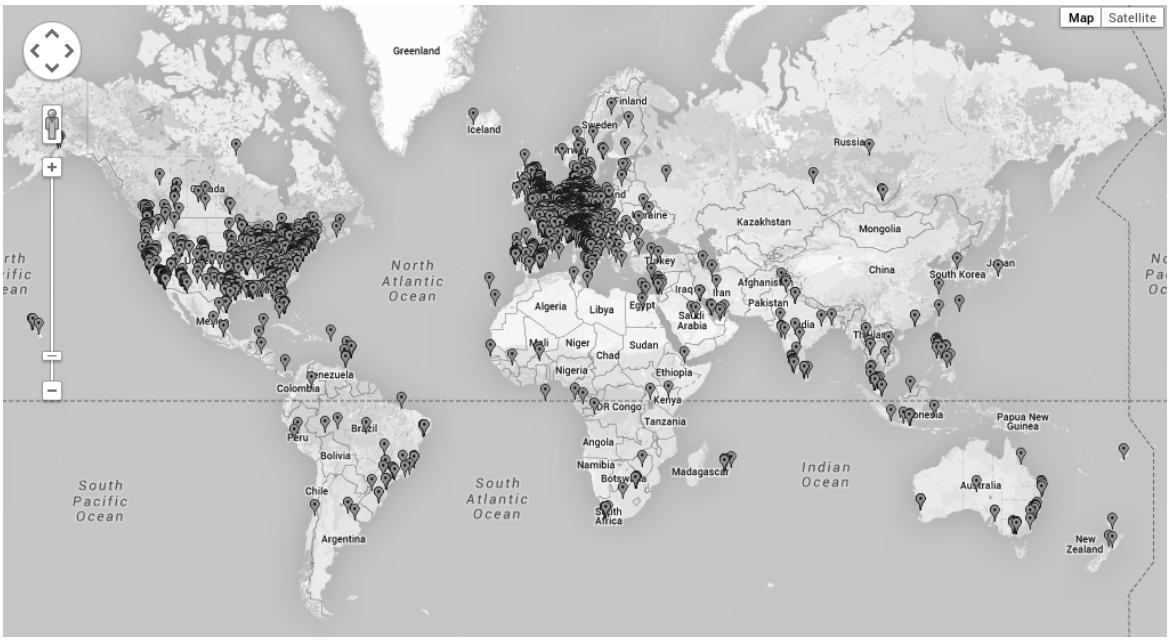


Figure 8.1: Geographical distribution of the Neubot clients in the world on the basis of their IP address. Figure refers to tests run in October 2013.

Example of DASH Simulation In this paragraph we show that our dataset not only provides information on the performance measured using the Dashtest rate-adaptation algorithm, but also allows one to drive simulation in which other rate-adaptation algorithms are assumed. In fact, the EAB is also a reasonable lower-bound approximation of the available bandwidth during the download of a segment (we say lower bound to account for the cases in which the Dashtest slows down). Therefore, assuming the available bandwidth constant during the download of a segment (i.e., during a time interval close to two seconds), one can linearly interpolate the values between EAB points to reconstruct the available bandwidth curve; in turn, this curve can be used to run simulations.

To demonstrate that the reconstructed EAB allows one to drive simulations involving varying rate-adaptation algorithms, we performed a simulation in which the reconstructed available bandwidth is used to simulate the behavior of three different rate-adaptation algorithms. To drive the simulation we used the results of a sixty-segments Dashtest performed over a mobile access network. We based our simulation on a mobile trace because, compared to fixed networks, the mobile has more-frequently-and-widely changing network conditions, therefore the resulting simulation allows us to evaluate more thoroughly the behavior of different rate-adaptation algorithms. After we computed the reconstructed available bandwidth, we used it to simulate the following three rate-adaptation algorithms: a) session average bitrate (SAB); b) last segment bitrate (LSB); and c) moving window average bitrate (WAB). SAB is the algorithm currently implemented in the VLC DASH plugin and requests the highest representation bitrate (HRB) which is lower than the average bitrate

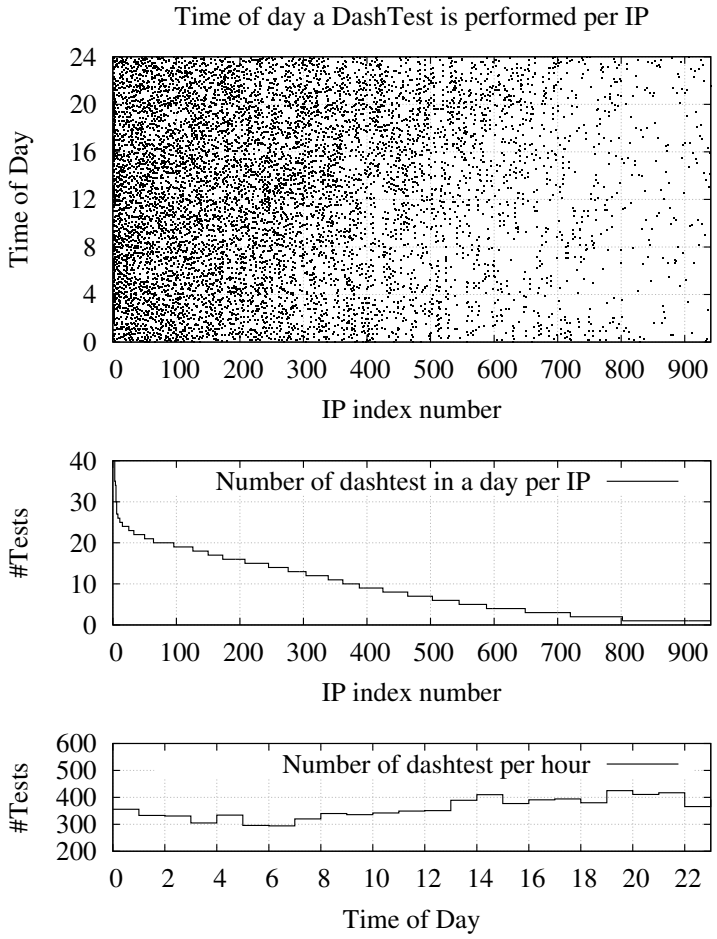


Figure 8.2: Distribution of Dashtests on November 2, 2013.

measured from the beginning of the streaming session. The main advantage of this algorithm is that the user does not experience changes in the video quality, however it does not adapt well to changing network conditions. As regards the LSB (of which the Dashtest rate-adaptation algorithm is a variation), it requests the HRB that is lower than the bitrate measured for the download of the last DASH segment. The main advantage of this algorithm is that it quickly adapts to the varying network conditions, however the video quality may change significantly as well. The WAB algorithm requests the HRB which is lower than the average bitrate measured over the last several video segments. As such, this algorithm is a compromise between the SAB and the LSB algorithms.

The simulation results (plotted in Figure 8.3) show that, predictably, the SAB algorithm is able to adapt to the varying network bandwidth only at the beginning of the streaming session: after a while the weight given to the past measurements is too high to allow for rate adaptation and, even in presence of bandwidth fluctuations, SAB is unable to change the requested representation bitrate. On the contrary the LSB algorithm presents, as expected,

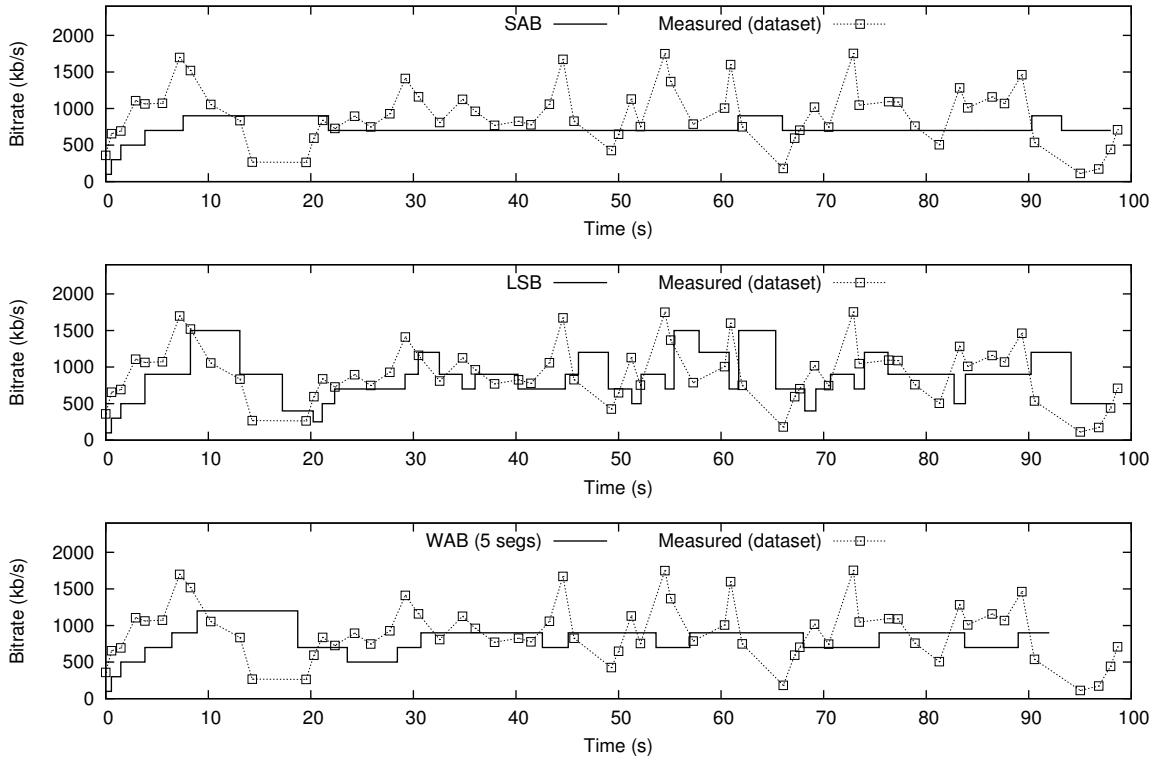


Figure 8.3: Simulation results. The simulation is based on the reconstructed bandwidth of a sixty-segments Dashtest experiment. Based on such data we simulated the bitrate of three common rate-adaptation algorithms.

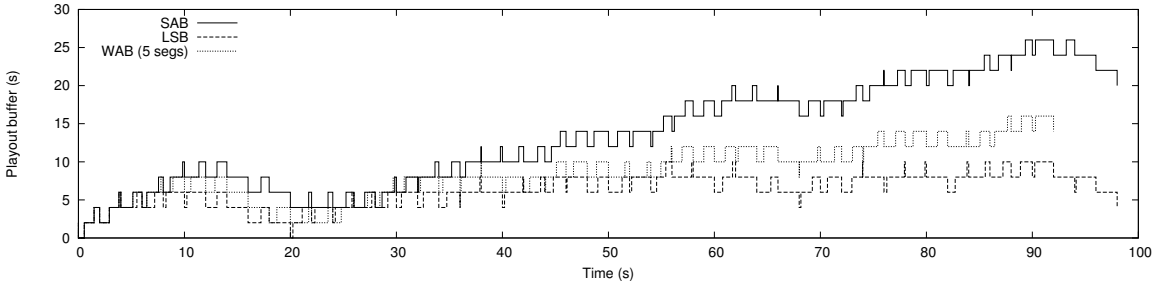


Figure 8.4: Simulation results. The simulation is based on the reconstructed bandwidth of a sixty-segments Dashtest experiment. Based on such data we simulated the playout-buffer level of three common rate-adaptation algorithms.

fast adaptation to the channel bandwidth with frequent and short changes in the requested representation bitrate. It is also worth noting that often the changes are towards a high quality representation; in fact, the overall video bitrate requested in the whole session is higher than in the SAB case. However, using SAB, the risk to incur in freezes during the video playback is high; in fact, the bottom graph in Figure 8.4 shows that, among the

tested adaptation algorithms, LSB is the one that keeps the playout buffer close to the minimum level. The WAB algorithm, finally, being a compromise between SAB and LSB, has predictably a slower rate-adaptation behavior than LSB and less stable video quality than SAB.

We conclude that the simulation was qualitatively correct, since the three simulated algorithms behaved as expected. However, it is not yet possible to conclude that the simulation was quantitatively correct. To do so, we will need to show that the simulated algorithms would behave according to the simulation under network conditions similar to the ones used to drive the simulation itself. To this end, we are planning to modify the Dashtest so that we can change the rate-adaptation algorithm used by the test and experiment with diverse rate-adaptation algorithms.

Relevance to Network Performance Measurements The DASH dataset also provides interesting insights from the point of view of network-performance measurements to increase the transparency of access networks. As the dotted line of Figure 8.3 shows, in fact, the Dashtest samples the network a number of times during a long download experiment, thereby providing information on the sustained performance that an access network can deliver. This kind of information is significantly different from the one provided by other Neubot network-performance tests (e.g., Speedtest), in which we estimate the maximum-achievable download speed in a short time frame (five seconds). Of course, when processing the results of the Dashtest to get insights into the sustained performance that an access network can deliver, one should keep in mind that the Dashtest modified-LSB rate-adaptation algorithm may reduce (let us simplify a bit) the download speed of the next chunk when the network conditions have worsened. However, it is not difficult to detect the cases in which the download speed was reduced: it suffices to find the samples for which the download time was higher than two seconds (the maximum expected download time). As part of our future work we plan to make the Dashtest more similar to a real DASH client to provide researchers and users not only with network performance insights, but also with information on the QoE that one could expect from video streaming on an a given access network.

8.4 Concluding Remarks

In this Chapter we presented the Neubot 'Dashtest', a network-performance test that emulates the DASH (Dynamic Adaptive Streaming over HTTP) MPEG standard. According to such standard, the media streaming is controlled by the client: the media file is divided into small chunks (or segments) encoded in multiple bitrate representations, and the client is responsible of selecting the most appropriate representation of each chunk, taking into account both the varying network conditions and the user quality of experience.

For selecting the most appropriate next-chunk representation, the Dashtest implements a modified last-segment-bitrate (LSB) rate-adaptation algorithm. According to LSB, the Dashtest requests the highest available representation that is lower than the speed at

which the last segment was downloaded. The algorithmic modification is the following: when the Dashtest detects that the network conditions worsened, it attempts to reduce the network usage proportionally to the difference between the download time and the expected download time (computed from the most recent download-speed estimate). This behavior is consistent with the Neubot promise that it will not consume too many network resources: if the user opens one-or-more bandwidth-hungry foreground flows, the Dashtest will detect that the network conditions worsened and will slow down, thereby releasing network resources.

The DASH dataset collected using Neubot is relevant for the community that studies media streaming, because the DASH standard does not specify the rate adaptation algorithm and because there are only few publicly available databases that can possibly be used as a basis to research novel rate adaptation algorithms. We showed that the data collected by the Dashtest can be used to simulate the behavior of diverse rate adaptation algorithms in the same network conditions experienced by Dashtest. A few examples provided a strong indication that the simulations were qualitatively correct; however, comparative experiments are needed to assess that the results are also quantitatively correct. To this end, we are planning to allow one to change the rate-adaptation algorithm used by Dashtest.

The Dashtest, moreover, is also relevant for our main research thread, i.e., experiments that aim at making access networks more transparent. In fact, the Dashtest probes the network with the kind of traffic (audiovisual streaming) that is reportedly the majority of consumer traffic. Moreover, the Dashtest is different from other Neubot tests (e.g., the HTTP speedtest) in that the Dashtest benchmarks the network performance with many repeated HTTP requests (rather than with a single HTTP request) and for a longer time (thirty seconds vs. five seconds). Therefore, it provides more insights on the sustained network performance that an access network can provide. To improve our benchmarking of access networks with respect to streaming, we are planning to enhance Dashtest to better emulate a real DASH client, to provide insights not only on the network performance but also on the quality of experience than the user can expect.

Finally, the Dashtest provides further evidence that the Neubot architecture is flexible and adequate for the purpose of collecting network-performance measurements from the edges. Not only, in fact, it was simple to develop and deploy the new Dashtest test, but also we were able to collect a large amount of data in a short time frame: just a few days after we released the version of Neubot including the Dashtest, in fact, Neubot was already performing (on the average) 360 Dashtests per hour.

Chapter 9

Experiment #4: BitTorrent Emulation

This Chapter is about the BitTorrent experiment, based on the namesake test implemented by Neubot, which emulates the BitTorrent peer protocol [196]. Such test was added to Neubot in the Spring of 2011, and the first Neubot version in which it was enabled by default was Neubot 0.4.0 (released on July 20, 2011). We added this test to benchmark the network using the BitTorrent protocol and, because the BitTorrent test methodology is similar to the Speedtest test methodology, to compare the BitTorrent test performance to the Speedtest test performance (as we shall see in Chapter 11).

This Chapter is organized as follows: in Section 9.1, we describe the BitTorrent test methodology; in Section 9.2, we explain that little analysis of BitTorrent data was performed within the doctorate (2011–2013), because we decided to focus on the packet-loss rate studies described in Chapter 7, and we refer the reader to Chapter 11 where the performance of the BitTorrent test are compared to the performance of the Speedtest test; in Section 9.3, we compare the BitTorrent test methodology to the Speedtest test methodology; finally, in Section 9.4, we draw the conclusions.

9.1 The BitTorrent Test Methodology

The BitTorrent algorithm does not differ significantly from the Speedtest algorithm used after Neubot 0.4.2 (see Section 6.7). In particular, both tests uses the closest M-Lab server. The main difference between the two algorithms is that the BitTorrent algorithm uses small messages (coherently with the way in which the real BitTorrent peer protocol works) while the Speedtest algorithm downloads a large message using a single GET request. Therefore, to efficiently use the bandwidth, the BitTorrent algorithm needs to pipeline many requests at the beginning of the test. In the following, we will describe version one of the test (there are three versions of the BitTorrent test, version one, version two and version three, and version one is the version that is run by default).

To understand the BitTorrent test methodology, it helps to know the semantic of the most-common peer-protocol messages. Those are described in the BitTorrent extension proposal (BEP) 0003 [196]. BitTorrent transfers pieces of data using the REQUEST and

the PIECE messages: the peer making the request sends a REQUEST message and the peer sending the response replies with a PIECE message containing the requested piece of data. According to the 2013-10-11 BEP0003 update, the maximum size that can be requested is 2^{14}; earlier versions of the specification, however, used the wording "[a]ll current implementations use 2^{15} (32KB), and close connections which request an amount greater than 2^{17} (128KB)"; therefore, we used 2^{17} for bandwidth efficiency. REQUEST messages contain the following parameters: the message *index* (within the vector of blocks that make up the file being shared), the byte from which to *begin* within the block, and the *length* of the piece. Similarly PIECE messages are characterized by an *index*, a *begin* offset and by the *piece* itself (i.e., the bytes). The INTERESTED message is used to tell to a peer that one would be interested to download data from it. A peer sends the UNCHOKE message to communicate the other peer that it can now make requests for data. Symmetrically, the CHOKE message tells to the peer that it is no longer allowed to make requests, and the NOT_INTERESTED message communicates to the peer that one is no longer interested to download data. The BITFIELD message is used at the beginning to tell the peer which chunks one has already downloaded. The HAVE message, not used by the BitTorrent test, is used to tell the peer that one has completed the download of one more chunk. The CANCEL message, not used by the BitTorrent test, cancels a REQUEST. The HANDSHAKE message is, as the name suggests, the first message exchanged by the BitTorrent protocol.

The protocol messages were originally transported using TCP; recent versions of the BitTorrent client (as well as of other clients), however, use instead the uTorrent Transport Protocol [259], which is a LEDBAT [283] implementation based on UDP. The BitTorrent test implemented by Neubot uses TCP; however, as of February 2014, we are currently developing an experimental test that uses the uTorrent Transport Protocol.

Figure 9.1 shows how we use the messages of the BitTorrent protocol to drive the phases of the test. Unlike the Speedtest test, the BitTorrent test does not include a phase in which the RTT is estimated at application-level. The 'connector' is the Neubot instance (henceforth, 'the instance') and the 'listener' is the test server. After the HANDSHAKE, the connector sends the INTERESTED message and the listener replies with an UNCHOKE message. Afterwards, the connector is allowed to download pieces from the listener, by requesting them with the REQUEST message. When the download is complete (more on this later), the connector signals the end of the download by sending the NOT_INTERESTED message. In turn, the listener replies with the INTERESTED message, to start the upload phase of the test. Upon receiving the INTERESTED message, the connector replies with an UNCHOKE message. Afterwards, the listener requests data from the connector using exactly the same algorithm used to implement the download phase, because the download and the upload phases are symmetrical. Once the download is complete, the listener sends the NOT_INTERESTED message, which terminates the BitTorrent test.

Algorithm 2 shows (using a pseudo Python language) the algorithm that implements the download phase of the BitTorrent test. Note that, as specified above, the download and the upload phases are symmetrical, i.e., during the download phase the connector is the Downloader and the listener is the Uploader, while during the upload phase the connector

Algorithm 2 BitTorrent test: the download algorithm

```
class UPLOADER
    method GOT_REQUEST(self, stream, index, begin, length)
        stream.send_piece(index, begin, random_data(length − begin))
class DOWNLOADER
    method INIT(self)
        self.saved_ticks ← 0
        self.inflight ← 0
        self.repeat ← 0
        self.complete ← 0
        self.generator ← None
    method GOT_UNCHOKE(self, stream)
        self.generator ← RequestsGenerator(β)
        for ρ in self.generator.get_initial_burst()
            stream.send_request(ρ.index, ρ.begin, ρ.length)
            self.inflight ← self.inflight + 1
    method GOT_PIECE(self, stream, index, begin, data)
        if self.saved_ticks == 0
            self.saved_ticks ← get_time()
        ρ ← self.generator.get_next_request()
        if ρ
            stream.send_request(ρ.index, ρ.begin, ρ.length)
            return
        self.inflight ← self.inflight − 1
        if self.inflight > 0
            return
        δ ← get_time() − self.saved_ticks
        if δ ≥ 1
            β ← β ∗ 5/δ
        else
            β ← β ∗ 2
        self.repeat ← self.repeat − 1
        if δ < 3 or self.repeat ≤ 0
            self.complete ← 1
            return
        self.saved_ticks ← 0
        self.got_unchoke(stream)
```

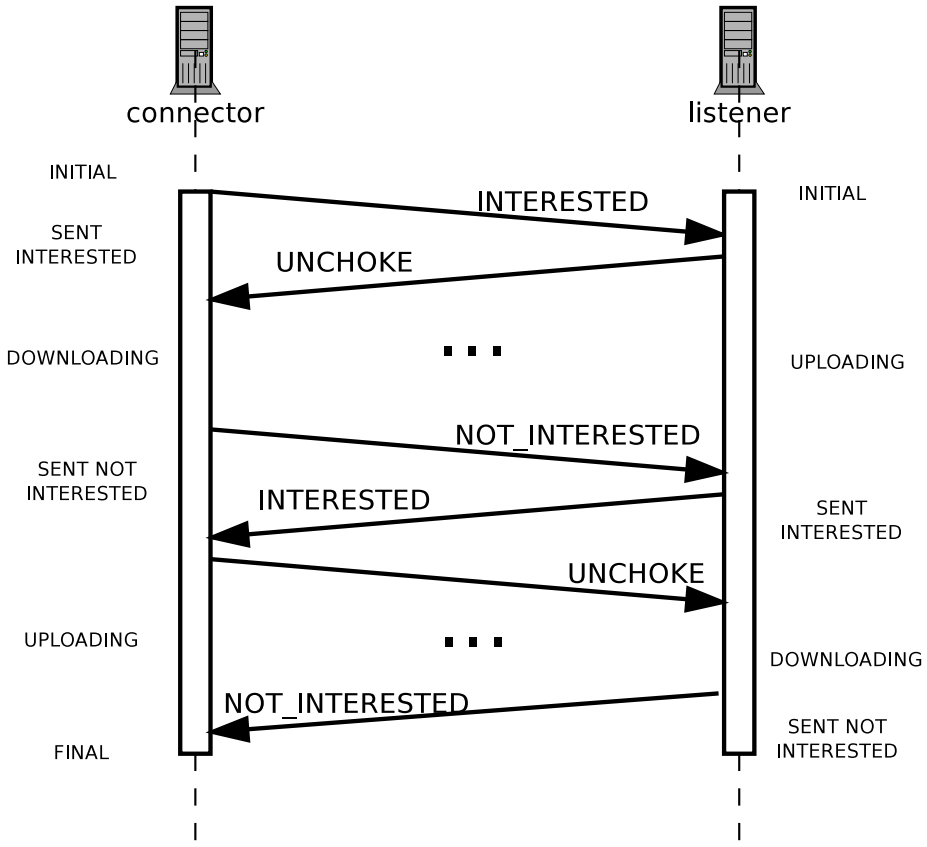


Figure 9.1: BitTorrent test overview.

is the Uploader and the listener is the Downloader. The Uploader is very simple: each time is receives a REQUEST, it sends the corresponding PIECE. The Downloader is much more complex, because it drives the download process. There is a global variable, β, that tracks the amount of bytes that the Downloader should request: at the outset β is set to 64000 and, as we will see, it is updated after each download. The *init* method is called once, to initialize the Downloader state variables. The download phase is complete only when the *complete* variable is set to nonzero. The *got_unchoke* method is called when the Downloader receives the UNCHOKE message: the Downloader creates a *RequestsGenerator* object, which is then used to generate a burst of requests, sent back-to-back to fill the space between the Downloader and the Uploader with BitTorrent messages. Note that the Downloader keeps track of the number of requests sent in the initial burst using the $inflight$ state variable. Moreover, note that the number of REQUEST messages in the initial burst (as well as the total number of REQUEST messages that will be sent) depends on the value of the β global variable, which is used to initialize the *RequestsGenerator*. The *got_piece* function is invoked when the PIECE message is received. When the first PIECE message is received, the Downloader saves the current time, which is used later to decide whether the test run for long enough. As long as the *RequestsGenerator* has more requests to send,

the Downloader sends a new REQUEST each time a PIECE is received. When there are no more requests to send, i.e., when ρ is empty, the Downloader waits until the pipeline is empty (i.e., when $inflight$ becomes zero). At that point, using an algorithm similar to the one used by Speedtest, the Downloader updates the β global variable, either linearly (if the download time, δ, is greater than one second) or exponentially (otherwise). Finally, the test is repeated if the download time, δ, is lower than three seconds and (unlike Speedtest) the download was repeated for less than seven times. Note that, to repeat the test, the code hackishly resets the relevant state variables and then calls the *got_unchoke* method of the Downloader.

The *RequestsGenerator* object works as follows: if requested to generate REQUEST messages that correspond to μ bytes of data, it inflates μ as follows: $\mu = \mu + 0.3 \cdot \mu$. The REQUEST messages corresponding to the first $0.3 \cdot \mu$ bytes are returned by *get_initial_burst* and are indeed used in the initial burst. The other REQUEST messages, instead, are the ones returned by *get_next_request*. The 'magic number' 0.3 was empirically determined: we noticed that this number allows to download at maximum speed in common ADSL and FastEthernet connections with typical RTTs.

9.2 BitTorrent Data Analysis

The BitTorrent experiment has been running since the release of Neubot 0.4.0 (June 20, 2011), and collected lots of data. The original plan was to perform comparative analysis of the BitTorrent and the Speedtest tests performance after the AICA 2011 paper [175]; however, as recounted in Chapter 6, after the AICA 2011 paper we become interested into estimating network-level characteristics (e.g., the packet-loss rate) from application-level measurements (see Chapter 7). Therefore, while we were working to the W-MUST [177] and CCR [178] papers, the comparative-analysis plan was effectively frozen.

We resumed the comparative-analysis plan in Summer 2013, as we shall see in Chapters 10–11, while we were working to the Neubot Visualizer (Neuviz). We anticipate the results of the comparison of the Speedtest and the BitTorrent tests performance: we found a few differences between the global performance of the two tests, which are to be investigated with more in-depth experiments. To prepare for the comparison of the BitTorrent and the Speedtest tests in Chapter 11, in the following Section we will summarize the main methodological differences between them.

9.3 Main Methodological Differences between BitTorrent and Speedtest

The main differences between the methodology of the BitTorrent test and the methodology of the Speedtest test are the following:

1) The BitTorrent test always measures the speed at the receiver, both during the download and the upload phases. Moreover, as we have seen, the BitTorrent test computes the

elapsed time as the difference between the time when the last byte of the last PIECE is received and the time when the first byte of the first PIECE was received. On the contrary, as detailed in Chapter 6, the Speedtest test measures the upload speed at the sender and, moreover, computes the elapsed time as the difference between the time when the last byte of the response is received and the time when the first byte of the request is sent.

2) The BitTorrent test needs to fill the space between the client and the server with a burst of messages sent at the beginning of the test, while the Speedtest test transfers a large number of bytes with a single GET (or POST) request.

9.4 Concluding Remarks

In this Chapter we described the BitTorrent experiment, based on the BitTorrent test, which emulates the BitTorrent protocol [196]. This test was added to Neubot to measure the network performance using the BitTorrent protocol. Within the Chapter, we described the BitTorrent test methodology, and we explained that we deferred the analysis of its data, to prioritize the work on the packet-loss-rate estimation (see Chapter 7). We also anticipated the comparative analysis of the BitTorrent and Speedtest tests data, performed as part of the Neuviz effort (see Chapters 10–11). To prepare such comparative analysis, we enumerated the main methodological differences between the two tests, which are the following: the BitTorrent test measures the upload speed at the receiver, while the Speedtest test measures it at the sender; unlike the Speedtest test, the BitTorrent test does not perform a long download (or upload) transfer, rather it attempts to emulate a long transfer by pipelining many REQUEST messages at the beginning of the transfer.

Chapter 10

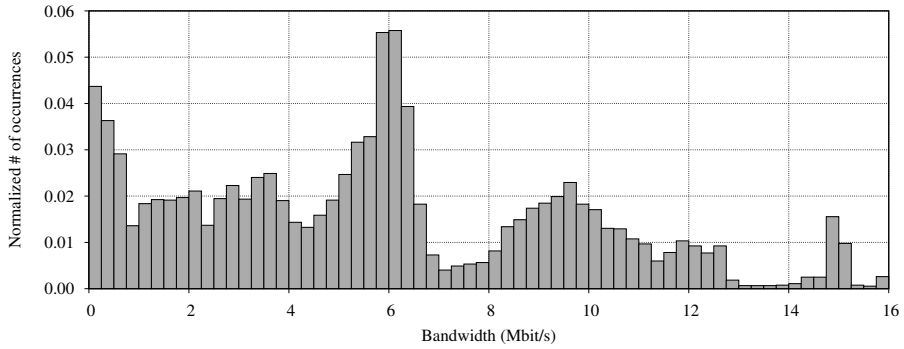
Neuviz: Towards Adapting the Testing Policy

In this Chapter we describe a system that will automatically process Neubot results to optimize the testing policy, e.g., by dynamically deciding which test should be run by which Neubot instance, taking into account the ISP and the geographic location of that instance. Because such system seems very complex, we isolated a subset of it, called Neuviz, designed to allow to navigate the results of Neubot tests on the world map and to spot anomalies to be mainly investigated manually. We describe the architecture and the first prototypical implementation of Neuviz, which in the next Chapter is evaluated by comparing the results of the BitTorrent and Speedtest data collected from January, 2012 to May, 2013. Based on our BiDATA 2013 [245], AICA 2013 [217] and Mondo Digitale 2014 [218] papers, this Chapter is organized as follows: in Section 10.1 we describe what motivated us to design Neuviz; in Section 10.2 we describe the architecture and the prototype of Neuviz; in Section 10.3 we draw the conclusions.

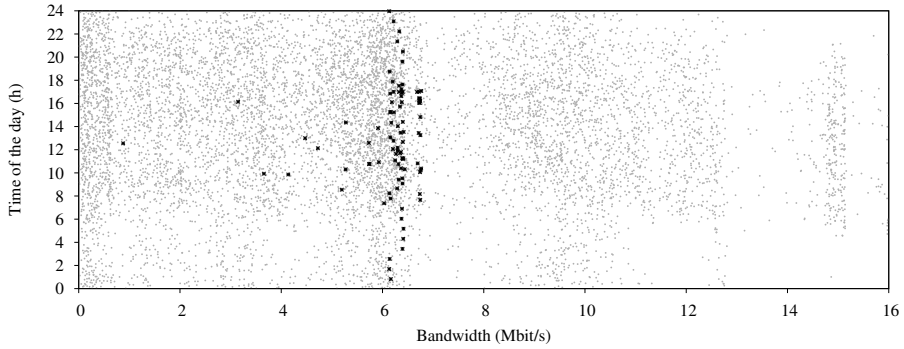
10.1 Motivation

One of the cornerstone ideas that characterize the Neubot architecture is the possibility of adapting the testing policy depending on the circumstances. For example, when a Neubot test shows an unexpected result, the Neubot instance should run more specific tests, trying to identify the specific cause of the anomaly. In this doctoral thesis, however, so far we have not dealt with either adaptation mechanisms or more-specific tests, because we concentrated on describing the software architecture and the measurements. However, recognizing that Neubot was mature enough, since Summer 2013, we have begun to study the design of a feedback loop by which the data collected by Neubot is processed to, in turn, influence the behavior of subsequent Neubot tests.

The idea of implementing a feedback loop that automatically allows a network-measurement tool to evaluate the collected data, possibly using coarse-grained scalable measurements as input, and to decide whether more specific tests should be run was described in



(a) Empirical download PDF



(b) Single user

Figure 10.1: Simulation: Automatic classification of user performance using already available data (simulation run using already-existing Neubot data).

literature by a number of papers, some of which are cited in the following. Of course, here we cannot avoid citing the seminal Neubot paper [201] written back in 2008 by Juan Carlos De Martin and Andrea Glorioso. A few years later, in their article for Science [262], Palfrey and Zittrain wrote: "[s]uggestions from the crowd might serve as hypotheses, automatically fed back into systems that in turn test them and release the results openly, creating a feedback loop." This approach is exemplified by the Herdict project [226], which aggregates web censorship information that anyone can contribute (using the Herdict web-based Reporter tool, the Herdict browser extension, as well as using email or Twitter messages) to create a 'web health map'. The collected data is used, in turn, to dynamically-update the list of possibly-blocked sites [62] that users can verify using the Herdict web-based Reporter tool.

Before reasoning on how Neubot can be evolved towards the direction envisioned by the 2008 seminal paper, let us assume that a feedback system is already implemented, to describe what can be done with such system to make access networks more transparent. In particular, let us assume that we already know the distribution of the download speed of users of a given ISP in a given geographic location, as shown in the example of Figure 10.1a. Let us also assume that the download speed distribution was computed not only using Neubot data, but also using data collected by other sources, e.g., an Azureus

plugin, such as DASU [278], and let us assume that the data collected by such plugin are periodically merged with Neubot data. The download distribution, especially if crowd-sourced by multiple tools, is of course already very informative, because it provides users with information on the performance that they should typically expect, should they choose that specific ISP in that specific geographic location. However it is even more interesting to show what can be done providing such information to a Neubot capable of using it to conduct more specific tests. Figure 10.1b shows the performance measured by a single Neubot instance (represented using black crosses) compared to the performance of all the instances (in gray). We can imagine that the Neubot instance has knowledge of the typical performance in its neighborhood, most likely limited knowledge but enough to pre-classify the results sent to the Neubot central server(s). In turn, the central server(s), using the instance pre-classification as well as its own knowledge, decides whether a measurement is in line with the other measurements performed in that area for that ISP, or whether there is a need for further investigating. In such case, the central server(s) could ask the Neubot instance to perform a longer network experiment in which, e.g., packets are captured and then sent to the central server(s) to be analyzed (automatically or manually). Finally, the automatic or manual analysis of the packet traces may lead the researchers to discover new, emergent behaviors for which new ad-hoc tests could be implemented and deployed.

Having briefly elaborated on what can be done with a Neubot that implements a feedback loop, let us now discuss what can be done to evolve the Neubot implementation in that direction. The first observation that we have is that we cannot use M-Lab servers for the purpose of implementing the central server(s) described above. The M-Lab servers resources, in fact, are all committed to assisting the server-side component of Neubot (and other tools hosted by M-Lab) to run the network experiments. Being Neubot data saved on Google Cloud Storage, and being some M-Lab data [1] already available on Google BigQuery[2], there is no doubt that part of the processing can be done using BigQuery. However, because the system described above assumes that the experiments results are pushed to the central server(s), and because instead the functionality allowing retrieve M-Lab data in real time as they are collected is not (as far as we know) planned for M-Lab, we are likely going to implement such a system on our own servers. In turn, this fact raises several issues regarding data scalability. Neubot, in fact, already generates a quite large amount of data per day (300 MB/day as of November, 2013), which is expected to grow as we add more tests and as we port Neubot onto mobile platforms. We can conclude that, to evolve Neubot into the dynamic system described in the above example, we must be prepared to deal with "big data" issues. In other words, we would better be prepared to scale, even though currently there is not need for us to scale yet.

To start off (and also because we needed a tool to study more thoroughly the performance difference between our Speedtest test and our BitTorrent test) we identified a

[1] Neubot data is not yet available on BigQuery, but the M-Lab team has committed to load on BigQuery the data of all the M-Lab tools.

[2] A web service allowing one to run SQL-like queries on Google-Cloud-Storage data.

subset of the system described above that could be already implemented. We call this subset 'semi-automatic feedback loop' because, in reality, there is no machine-backed automation. Rather, this simplified system is a system that has a web user interface that allows us, researchers working on Neubot, to browse the available (Speedtest and BitTorrent) data to search for anomalies worth studying. Once such anomalies are identified (by us rather than by a machine) we will look for the most rational way to investigate them more in depth. This process will hopefully allow us to understand the reason of the anomalies, be it the behavior of the network or our measurement code, leading in turn to a better understanding of the network and/or to better tools. More specifically, in the rest of this Chapter, we will describe Neuviz (the Neubot visualizer), a software architecture, designed for scalability, that implements such semi-automatic feedback loop and that already includes some key features on top of which it should be possible to implement more feedback automation.

10.2 Neuviz: the Neubot Visualizer

In this Section we present Neuviz, a data processing and visualization architecture for network measurement experiments, which has been tailored to work on Neubot data. In particular, Neuviz was designed to implement the semi-automatic feedback loop that we described in the previous Section. In this Section we limit our discussion to the architecture of Neuviz and to the implementation of the Neuviz prototype. In the following Chapter, we will evaluate Neuviz, using it to compare the results of the Neubot BitTorrent and Speedtest tests.

10.2.1 Description of the Neuviz Architecture

Figure 10.2 shows the Neuviz architecture, which is a pipeline that processes data provided by Producers, and which organizes the data such that Consumers can visualize (or further process) such data. The pipeline is composed of a Backend and a Frontend: the Backend receives data from many Producers and processes such data to allow for efficient visualization; the Frontend is a Web interface that visualizes the data. In the middle there is a Web API. Note that the Web API can also directly access the Raw Database: direct access makes sense when one can efficiently process the data on the fly using the Web API (and is also handy when one wants to experiment with a new kind of Analysis without taking the burden of writing an Analysis Stage first).

The Producers

As a first approximation a Producer is a static dataset. For example, here we used Neubot data expressed in CSV format and in the future we may want to import datasets from other projects (e.g., Speedtest.net [136]) and/or encoded in other formats (e.g., JSON). Neuviz also includes a Submit API, which allows tools (mainly Neubot but possibly also other network-measurement tools) to push the result of their experiments just after the experiments are completed. We added the Submit API because, as mentioned above, we want to

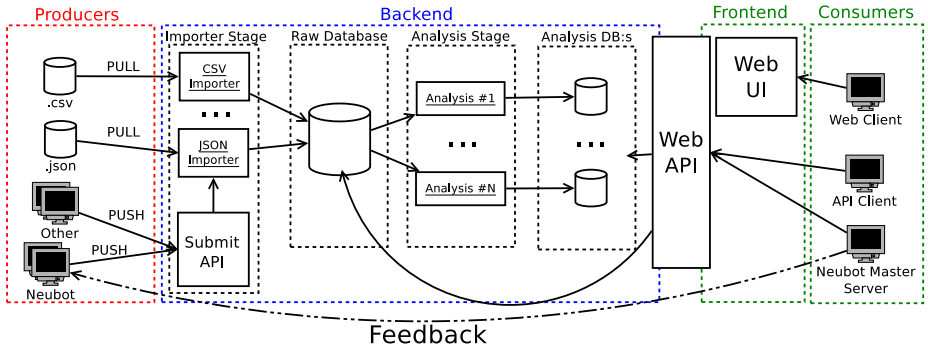


Figure 10.2: The Neuviz software architecture.

create, in the long term, a feedback loop in which the data posted by Neubot is processed by Neuviz and consumed by the Neubot master server(s), which will automatically provide better suggestions to Neubot instances.

The Backend

The Backend receives network-experiments data from many sources and organizes such data for an efficient visualization. As Figure 10.2 shows, the Backend is composed of two processing stages, each followed by a database stage. The first processing stage is the Importer Stage, which receives data from many sources, normalizes the data, and writes the data into the Raw Database. The second processing stage is the Analysis Stage, which reads data from the Raw Database, analyzes the data to compute aggregate metrics, and saves the aggregate metrics into one or more Analysis Databases. In the following Paragraphs we discuss the stages of the Backend, starting from the Importer Stage.

The Importer Stage The Importer Stage organizes data coming from many sources, and possibly represented using different formats, into a single database. There is an Importer Module for each network measurement tool and data format. To make an example, if we want to use Neuviz to visualize Speedtest.net data (expressed in CSV format) and Neubot data (expressed in CSV and JSON format), we need to write three Importer Modules: one for the Speedtest.net data and two for the Neubot data (the former for the CSV format and the latter for the JSON format). The Submit API design reflects the fact that there is an Importer Module for each network-measurement tool and data format. The basic API request to store the result of a new experiment, in fact, is a POST request to this URI: '/neuviz/1.0./import/tool/params', where tool is the name of the tool that produced the piece of data (e.g., 'neubot') and where the 'Content-Type' HTTP header reflects the data MIME type (e.g., 'application/json'). The problem of whether (and how) to authenticate the measurement tool submitting the data is not discussed here. The Importer Stage does not reduce all the input data to the same schema (be it a real SQL schema or not), because such transformation is not practical. The input data schema, in fact, depends on which metrics the specific network experiment measures; therefore, this stage only performs the

following tasks: it validates the input data, it enriches the data with geographical information (if needed), it converts the data into a common, database-dependent format (e.g., JSON), and writes the result into the Raw Database.

The Raw Database The Raw Database receives heterogeneous data organized in a uniform format (e.g., JSON) by the Importer Stage. As said before, it is not practical to reduce all the input data to the same schema, suggesting that the Raw Database could be easily implemented using NoSQL (e.g., MongoDB [90]). A possibly-conflicting requirement is that, in addition to being able to store heterogeneous data, the Raw Database shall also be scalable-enough to handle continuous streams of data posted on the Submit API by, at least, Neubot and possibly by other network measurement tools.

The Analysis Stage The Analysis Stage is a collection of Analysis Modules that periodically fetch data from the Raw Database and process it to produce the aggregate data needed for the visualizations. We plan to implement two different visualizations[3]: one that shows a given performance metric (e.g., the median download speed) on the world map and that allows the user to zoom and see the same performance metric on a smaller geographic scale (i.e., country, province, city); the other that shows a given performance metric in function of the time.

As far as functional requirements are concerned, the Analysis Stage needs to process data in a scalable way, because we need to process multiple times the raw data stored in the Raw Database. Also, the Analysis Stage should minimize the computational cost of adding the results of new experiments to Neuviz.

The Analysis Databases The Analysis Databases are a number of (conceptually-separated) databases that store data which is ready to be visualized on the Neuviz Frontend with minimal computational cost. We want, in fact, to allow the user to visualize and browse the data as seamlessly as possible.

The Web API

The Web API connects the Backend and the Frontend. The Frontend, in fact, uses the Web API to retrieve the data to be visualized by a Web client through the Neuviz Web interface. However also other clients can access the Web API to extract information from the collected data. The Web API typically returns the Analysis Database data, because Neuviz is optimized to store and quickly return the results of the data analyses. However, in cases in which the cost of processing the Raw Database data on the fly is negligible, the Web API will access directly the Raw Database data and will compute the result on the fly. This is represented in Figure 10.2 by an arrow that goes from the Web API to the

[3]At the moment of writing this doctoral thesis, the former visualization is implemented, the latter is not yet implemented.

Raw Database. In this Chapter we do not discuss whether and how the access to the API should be restricted. This could be the subject for a future work.

The Frontend and the Consumers

The Frontend is a Web interface that visualizes the data stored in the Backend. The typical (and default) Consumer is of course a Web client that uses the Neuviz Web interface, but also other clients could consume the available data. In particular, as mentioned above, in the future we plan to let the Neubot master server(s) retrieve data from the Web API, to process such data and accordingly adapt the suggestions provided to the Neubot instances. For example, if there are few Neubot instances in a specific geographical area, the Neubot master server(s) will suggest to perform tests more frequently in that area.

10.2.2 Data Used in the Neuviz Prototype

To evaluate the first Neuviz prototype we imported the results of the Speedtest and of the BitTorrent Neubot tests collected by all Neubot servers in the January 2012 – May 2013 period. The evaluation is performed in the following Chapter; however, to understand the following Subsection and, in particular, Figure 10.3, it is important to explain the meaning of the data loaded into the Neuviz prototype.

The following are the most relevant fields saved by the Speedtest and the BitTorrent tests:

IP addresses The results of both tests include both the IP address of the Neubot instance and the IP address of the server. The IP address of the instance is the address reported by the server, which is known for sure to be public, just in case the instance was behind a network address translator (NAT).

Timestamp The timestamp is the time in which the test was run according to the server, measured as the number of elapsed seconds since January 1, 1970 UTC.

Neubot ID The Neubot ID (also called 'instance ID' or 'UUID') is a random unique identifier that uniquely identifies a Neubot instance.

Connect time The connect time is an upper-bound estimate of the minimum RTT of the end-to-end path between the Neubot instance and the server. It is computed as the difference between the time in which the socket is reported to be actually connected [183] and the time in which the connect() system call returned.

Download and upload speeds The speeds are computed dividing the number of received (or sent) bytes by the elapsed time.

Having defined the most relevant fields, let us describe the first Neuviz prototype and explain our implementation choices.

10.2.3 Description of the Neuviz Prototype

In this section we describe the implementation of the first Neuviz prototype [114], and we explain our implementation choices.

The Importer Stage The Importer Stage is a Python command-line script that accepts as input a CSV file. In our tests we imported and normalized 1.5 GB of Neubot data (using CSV files), from January 2012 to May 2013, and we stored the data into a MongoDB database. We ran the code on a laptop with an Intel Core i7 CPU at 2.0 GHz, with 8 GB of RAM, and a 256 GB SSD, running GNU/Linux 3.5.0. The Python code is designed to execute both on a common computer and in a cloud environment, if needed: to this end we divided the Importer and the Analysis code into a *map* step and a *reduce* step. We also used the GeoLite Free Database [11] to retrieve geo-information from the client IP address using MongoDB to store the geographic information. As explained in the GeoLite website, when the database is not up-to-date, the geolocation loses 1.5% of accuracy each month because IP addresses are re-assigned. To minimize the damages caused by out-of-date GeoLite databases, we never used databases older than two months.

The Raw Database We implemented the The Raw Database using MongoDB, a NoSQL database often deployed in big data scenarios [256]. We exploited the MongoDB indexes feature to speed up the query execution, processing about 5.3 million of samples in less than 60 minutes. As noted above, the code is written in a way that potentially allows us to use MapReduce on cloud services (e.g., using Hadoop [61]), should we need to do that. However, during the development of the initial prototype, we did not use MapReduce, because a single NoSQL database allowed us to perform queries on demand and retrieve data immediately (which is not possible in a cloud-based MapReduce scenario).

The Analysis Stage We implemented a prototypical Analysis Module, written in Python, to retrieve and process data from the MongoDB database to create the visualizations that allow us to navigate Neubot data. The Analysis Stage that we implemented outputs a JSON file in which the information is aggregated at the geographical level (countries, and cities), at the temporal level (hour of the day), and at the network-access level (ISP). Therefore, the Web interface receives in input, for BitTorrent and Speedtest: the median value of the upload speed, of the download speed, and of the connect time, organized by country city, ISPs, and hour of the day. We decided to use the median, which is a common index used to analyze network traffic, to avoid the risk that few outliers would inordinately skew our index. We also computed the number of Neubot instances (per country, city, ISP) as well as the number of Neubot tests (per country, city, ISP). Since the IP address can vary over time, we identified a Neubot instance by using the (Neubot ID, IP address) tuple. The number of instances and the number of tests can be used to understand the geographical distribution of Neubot clients and the network traffic produced by each Neubot.

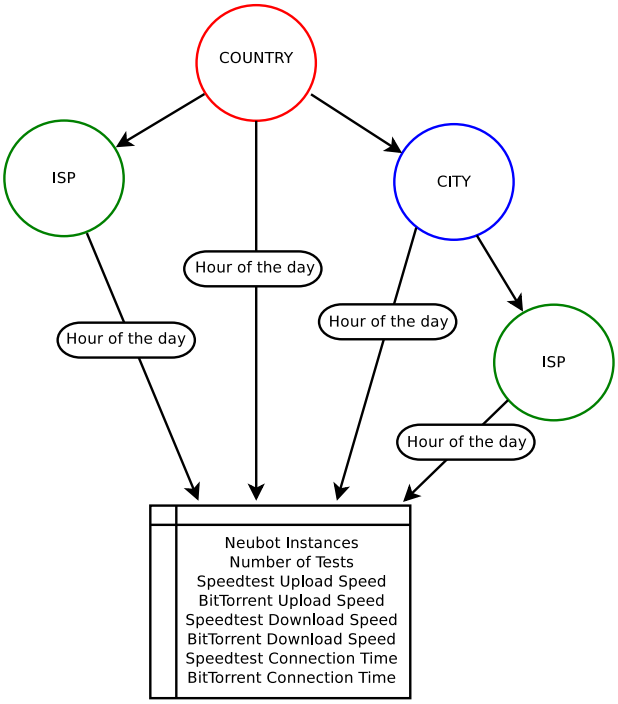


Figure 10.3: Representation of geographical (country, city), time (hour of the day), and network-access dimensions (ISP) of a JSON file.

The Analysis Databases For each month of data stored in the Raw Database, we generated a JSON file using the prototype Analysis Module. The collection of these JSON files can be considered to be the Analysis Databases. A frontend, written using Node.js [118], is currently being developed to serve such JSON files, as well as to serve more advanced requests in which finer- or coarser-grained data is requested. The JSON files are currently stored on disk, for implementation simplicity; however, in the future they will most likely be stored into a MongoDB database, to simplify the task of retrieving finer- and coarser-grained data. Moreover, when the volume of data will increase, we will have the option to store the data in the cloud, if needed.

The Web API To access the Neuviz API, the user sends the following HTTP/1.1 request: 'GET /neuviz/1.0/viz/params', where viz is the name of the visualization, and params is a placeholder for (possibly-empty) parameters. As anticipated above, the API is being implemented using a frontend written in Node.js. The returned JSON contains a recursive set of dictionaries that represent the geographical dimension (country, city), the time dimension (hour of the day) and the network-access dimension (ISP). The leaves are dictionaries that contain the following hour-wide median statistics for the Speedtest and the BitTorrent tests: download speed, upload speed, connect time, number of Neubot instances, number of tests. The geographical (country, city), the time (hour of the day), and the business (ISP) dimensions of data is shown in Figure 10.3.

The Web Interface The Web interface, written using D3.js [40], allows the user to explore different network measurement performances at different geographic dimensions (country, cities, and ISPs). In particular, the Web Interface allows one to navigate the BitTorrent and Speedtest median performance of each country and of each ISP on the world map. For simplicity, and since it does not seem to cause any performance issue, we currently use the Web interface to compute some statistics, e.g., the difference between the median Speedtest download speed and the median BitTorrent download speed that we will use in the next Chapter to compare the performance of BitTorrent and Speedtest.

10.3 Concluding Remarks

We opened this Chapter with a description of a system to organize the data collected by Neubot and by other tools, to provide information on the network-performance characteristics of the Internet, and, more importantly, to allow Neubot to adapt its testing policy depending on the circumstances (which has been one of the key Neubot design goals since the inception). We also explained why it is not simple to implement such a system only using the facilities kindly made available by M-Lab to researchers working on making access network more transparent. Therefore, because we are committed to design and implement such a system, we described a subset called Neuviz, for 'Neubot visualizer', designed to navigate the Neubot results to find out network anomalies to be further investigated (manually, for now).

Afterwards, we described the architecture of Neuviz, which is organized as a pipeline that receives data in input either from static files or from a specific Web API. The incoming data is preprocessed, normalized, enriched with geolocation data, and stored into the so-called Raw Database, which contains the whole corpus of Neubot data. Next, we described the Analysis Modules, modules that periodically process portions of the Raw Database to compute statistics that are saved into the more specific Analysis Databases. In front of such pipeline there is a Web API that allows one to query the available data. Depending on the cost of processing (and on the programmer's convenience) the Web API pulls data either from the Analysis Database or from the Raw Database (in the latter case, of course, the Web API also needs to compute statistics on the fly). The Web API is coupled with a default Web Interface, to browse the data.

We implemented the first Neuviz prototype using a Python-based backend, in which the processing is already organized into a *map* and a *reduce* stages, allowing us to scale to the cloud if/when the available data will become considerably larger than it is now. The Web APIs (both the one to submit new data and the one to query the Databases) are instead written using Node.js, and the Web interface is based in D3.js. To validate Neuviz, we loaded into the prototype one year and a half of BitTorrent and Speedtest Neubot data, and we programmed the Web interface to navigate the BitTorrent and Speedtest median performance of each country and of each ISP on the world map. In the next Chapter we show how we used Neuviz to compare the results of the BitTorrent and Speedtest tests.

Chapter 11

Neuviz Evaluation: BitTorrent vs. Speedtest

In this Chapter we show that Neuviz is an effective tool to navigate Neubot data to identify cases (to be investigated using more specific network tests) in which a protocol seems discriminated. To this end, we load into Neuviz the Speedtest and BitTorrent results collected by all Neubot servers in the January 2012 – May 2013 period, and we visually investigate such data looking for anomalies. Based on our AICA 2013 [217] and Mondo Digitale 2014 [218] papers, this Chapter is organized as follows: in Section 11.1 we start by explaining our motivation; in Section 11.2, we briefly recap the methodology of the BitTorrent and the Speedtest tests; in Section 11.3, we recap the format of the BitTorrent-and-Speedtest data used for evaluating Neuviz; in Section 11.4, we evaluate Neuviz by investigating the data and looking for anomalies; finally, in Section 11.5 we draw the conclusions.

11.1 Motivation

There were two synergic reasons behind our decision to evaluate Neuviz using the Speedtest and the BitTorrent data. On the one hand, of course, we wanted to validate the tool that we have designed and implemented. On the other hand, we also wanted to start a research thread in which we compare the performance of the Speedtest and the BitTorrent tests, to spot interesting anomalies and/or to improve our testing methodology.

11.2 Description of BitTorrent and Speedtest

For the reader's convenience, in this Section we briefly describe the methodology of the BitTorrent and the Speedtest tests, and we also provide pointers to more detailed descriptions. We start this Section by describing which flavor of Speedtest we used for the comparison (more info on the evolution of the Speedtest test in Chapter 6). Next, we describe the methodology of the BitTorrent test (see Chapter 9). Finally, we dedicate a

paragraph to remind the most significant differences between the two test methodologies (see Section 9.3).

The Speedtest Test The Speedtest test flavor used in this evaluation of Neuviz is the one described in Section 6.7: one connection is used and the operating system is free to manage the receiver's buffer. In other words, the Speedtest flavor used here was functionally equivalent to Algorithm 1 with $\sigma = 0$ and $\nu = 1$. Also, let us remark that the Speedtest takes place with the closest M-Lab server.

The BitTorrent Test The BitTorrent algorithm (covered in detail by Section 9.1) does not differ significantly from the Speedtest algorithm described in Section 6.7. In particular, both tests uses the closest M-Lab server and both use a single TCP connection.

Methodological Differences Between BitTorrent and SpeedTest The main methodological differences between the two tests are the following: the BitTorrent test measures the upload speed at the receiver, while the Speedtest test measures it at the sender; unlike the Speedtest test, the BitTorrent test does not perform a long download (or upload) transfer, rather it attempts to emulate a long transfer by pipelining many REQUEST messages at the beginning of the transfer.

11.3 Data Used for the Evaluation

The evaluation is performed using the results of the Speedtest and of the BitTorrent Neubot tests collected by all Neubot servers in the January 2012 – May 2013 period. The most relevant fields saved by two tests were already described in Section 10.2.2; for the reader's convenience, the same description is also provided below.

IP addresses The results of both tests include both the IP address of the Neubot instance and the IP address of the server. The IP address of the instance is the address reported by the server, which is known for sure to be public, just in case the instance was behind a network address translator (NAT).

Timestamp The timestamp is the time in which the test was run according to the server, measured as the number of elapsed seconds since January 1, 1970 UTC.

Neubot ID The Neubot ID (also called 'instance ID' or 'UUID') is a random unique identifier that uniquely identifies a Neubot instance.

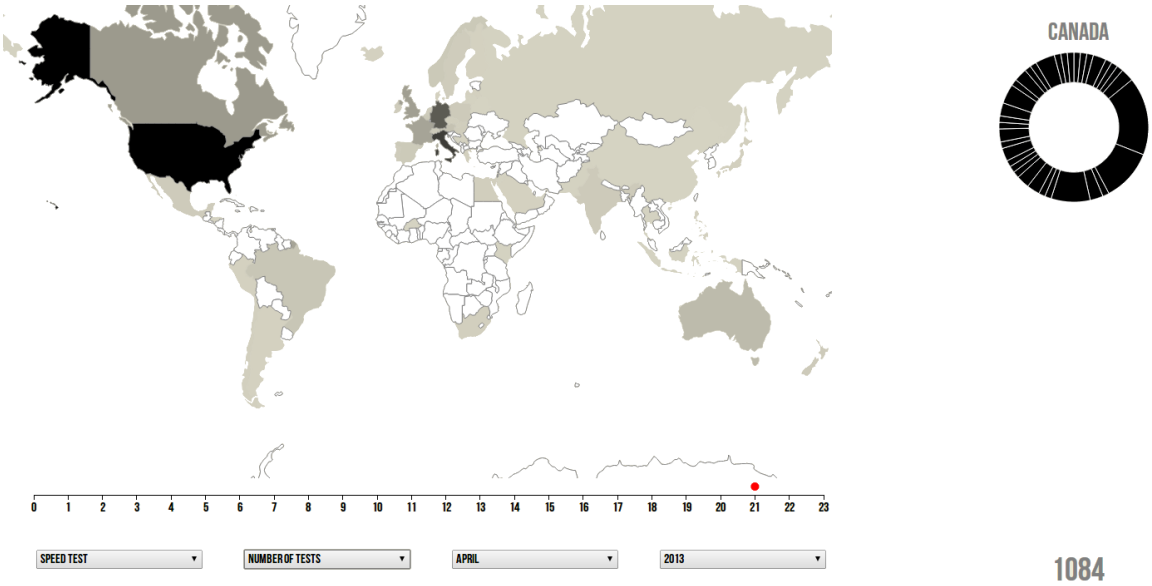


Figure 11.1: Neuviz interface showing April-2013 Neubot data on the world map.

Connect time The connect time is an upper-bound estimate of the minimum RTT of the end-to-end path between the Neubot instance and the server. It is computed as the difference between the time in which the socket is reported to be actually connected [183] and the time in which the connect() system call returned.

Download and upload speeds The speeds are computed dividing the number of received (or sent) bytes by the elapsed time.

11.4 Evaluation of Neuviz

This Section contains a preliminary evaluation of the Neuviz prototype described in the previous Chapter. We start by showing how Neuviz can provide a map of the number of Neubot tests and instances per country and per Autonomous System. Next, we evaluate Neuviz by looking at BitTorrent and Speedtest data, in search for anomalies.

11.4.1 Number of Neubot Tests

Figure 11.1 shows the visualization of the number of tests per country and per hour. The alpha channel of the country color indicates the median number of tests per country (the more opaque the color, the most tests in such country). The visualization, in particular, shows the median number of tests performed between 9:00 PM and 10:00 PM (local time) in April 2013. The selected country is Canada, in which the median number of tests performed is indicated by the number in the bottom right corner (1,084). By selecting other countries

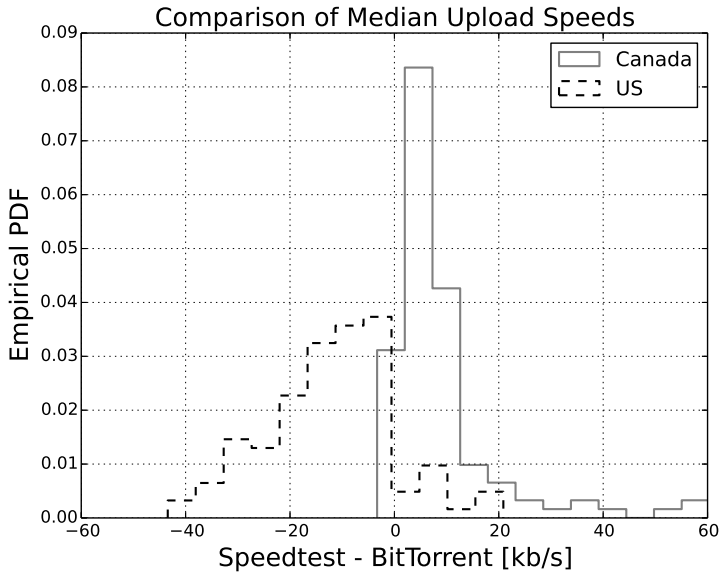


Figure 11.2: Empirical probability density function (PDF) of the difference of the median upload speed of US and Canada.

in the visualization, we have seen that the countries with more median tests per hour between 9:00 PM and 10:00 PM in April 2013 are: the US (4,223); Italy (2,866); Germany (2,285); and Canada (1,084). The availability of the number of tests per country is interesting because, by knowing the number of tests per country, the Neubot master server(s) could optimize the test coverage; e.g., if in a country there are few Neubot instances, such instances could be instructed to perform more tests per hour, thereby increasing the number of performance samples available for such country.

11.4.2 Comparison of Speedtest and BitTorrent Performance

Before studying the visualization that shows the difference between the Speedtest and the BitTorrent test download and upload speeds, we checked whether the Speedtest and the BitTorrent connect times were 'comparable'. To this end we arbitrarily defined 'comparable' two median connect-times whose difference was smaller than five millisecond, in our experience a reasonable threshold for comparing connect times.

The visualization of the difference between the median all-ISPs BitTorrent connect time and the median all-ISPs Speedtest connect time shows, surprisingly, that in Italy such difference is always positive and often greater than five millisecond (i.e., the Speedtest connect time is typically lower than the BitTorrent one). Italy is the only country in which, for 2013 data, we noticed this behavior. It was also interesting the comparison of the median upload speed in countries in which the median connect times are comparable. We noticed, e.g., that in 2013 the median upload difference between Speedtest and BitTorrent in Canada was very often positive, while the same difference was very often negative in

the US (see Figure 11.2). When comparing the download speeds of countries in which the connect times are comparable, we also noticed that the US Speedtest download speed is always lower (in median) than the BitTorrent download speed for every hour of the day and for every month of 2013. Interestingly, instead, the download speeds are comparable in Italy, in which, however, as we have seen, there is a connect-time bias in favor of Speedtest.

The above observations lead us to speculate that: (a) the implementation of the BitTorrent test is such that it is, in general, slightly faster than Speedtest; (b) in Italy, however, the two tests have comparable median download speeds, possibly because the connect-time bias that we observed rebalanced the speed bias in favor of BitTorrent; (c) the BitTorrent upload speed seems to be discriminated in Canada. Of course, these are only hypotheses that need to be verified (or contradicted) by more detailed experiments (e.g., experiments in which we capture packets).

To wrap up, Neuviz allowed us to discover three specific network anomalies. In the future, more advanced master server(s) could learn, from the Neuviz API, about similar anomalies and ask Neubot instances that are near the anomalies to gather more information (e.g., one could capture packets to gather RTT samples useful to understand whether there is indeed a connect-time bias).

11.5 Concluding Remarks

In this Chapter we evaluated Neuviz by using it to browse one-year-and-a-half records collected by two network tests periodically run by Neubot, Speedtest (based on HTTP) and BitTorrent. We are motivated to do so, not only because we wanted to evaluate Neuviz but also because we wanted to compare the Speedtest and BitTorrent performance, to find anomalies and/or to improve our testing methodology.

Before evaluating Neuviz using Neubot data, we briefly reviewed the Speedtest methodology and the BitTorrent test methodology. Afterwards, we showed that Neuviz effectively helped us to identify cases (to be investigated with more specific network tests) in which a protocol seems discriminated. As anticipated in Chapter 10, we plan to extend Neuviz to automatically raise warnings, thereby helping us to detect anomalies.

Chapter 12

Conclusion and Future Work

In this thesis we described the Neubot (Network neutrality bot) software, a client-side software platform for distributed network performance measurements that emulates several protocol with the aim of increasing network transparency. We started with a description of network neutrality (Chapter 1) because Neubot is designed to collect data useful to study network neutrality. Afterwards, we described related work (Chapter 2), in particular tools that measure network performance and that aim at increasing the network transparency, by revealing traffic discrimination and shaping.

Next, we described the Neubot architecture (Chapter 3), which is one of the main contributions of this thesis. Neubot is a tool that runs in the background and periodically performs active network-performance tests with either test servers (client-server mode) or other Neubots (peer-to-peer mode). Neubot tests, which emulate diverse application protocols, are implemented as plugins that can be updated independently of the Neubot core. Neubot can adapt the testing policy (i.e., which test to run and with which server/peer) depending on the ISP and on the geographic location in which a Neubot instance is. Neubot can run arbitrarily-long measurements because it monitors the hosting computer load to understand when it needs to release system and network resources.

Subsequently, in Chapter 4 we described the current implementation of Neubot (referring to Neubot 0.4.16.9). Although some advanced features (peer-to-peer tests, arbitrarily long tests, automatic adaptation of the testing policy depending on the ISP/location) were not yet implemented, Neubot proved to be already flexible enough to allow us to run the four large-scale network-performance-experiments campaigns described in Chapters 6–9.

In Chapter 5 we recounted the most relevant events in the Neubot history and also provided up-to-date Neubot numbers (e.g., the number of performed tests/day).

The following three Chapters described four measurements campaigns. These campaigns were only possible because more than 1,000+ users installed Neubot and because Neubot allowed us to easily deploy new experiments on such Neubot instances.

More specifically, in Chapter 6 we described the 'HTTP speedtest' campaign, run to measure the broadband speed of the Neubot users. In such Chapter we also describe the evolution of the 'HTTP speedtest', influenced by further studies and interactions with other researches. In fact, the 'HTTP speedtest' was originally designed to estimate the

broadband speed, just like the Speedtest.net test, and was later modified to more sensitive than originally to the quality (and, in particular, the losses) of an end-to-end path.

In Chapter 7 we studied the link between the goodput (i.e., the application-level speed) and the packet loss rate (PLR), a key parameter to understand the TCP performance. This study of the PLR is the other major contribution of this thesis and is – in perspective – one of the main building blocks needed to allow the class of applications like Neubot to implement distributed network-neutrality measurements. By linking a certain application-level measurement with its probable network-level cause, in fact, one not only documents the behavior of the network (e.g., protocol X seems to be discriminated), but also explains what happens (e.g., protocol X consistently experiences a high packet-loss rate).

In Chapter 8 we use Neubot to study the performance of the Dynamic-Adaptive-Streaming-over-HTTP (DASH) streaming technology. Motivated by the desire to collect a distributed dataset useful to study DASH rate-adaptation algorithms, this study is not only interesting for the research community but also from the point of view of network transparency, being multimedia streaming one of the most widely used Internet applications (e.g., YouTube and Netflix use DASH to deliver video).

In Chapter 9 we described the methodology of the BitTorrent test, which emulates the BitTorrent protocol. This test was added to Neubot to measure the network performance using the BitTorrent protocol, and, because the methodology used by this test is similar to the one of the HTTP Speedtest test, to compare the performance measured by the BitTorrent test to the one measured using the HTTP Speedtest test. An initial comparison of the performance of the two tests is presented in Chapter 11.

We conclude the thesis with Chapters 10 and 11, in which we describe Neuviz, one of the Neubot-related aspects on which we are currently working. Neuviz is an architecture for data processing and visualizations that imports the Neubot data and allows us to navigate such data in the Web browser, looking for anomalies. In due time, we expect Neuviz to automatically provide the statistics needed to automatically adjust the Neubot testing policy mechanism. So far, despite Neuviz being only a prototype, it already allowed us to find three specific anomalies concerning the distribution of the median values measured by two Neubot tests ('HTTP Speedtest' and BitTorrent) in different countries. Such anomalies are now to be investigated with more specific tests to understand whether they are caused by our testing methodology or by the network.

Speaking of future work, the research and development of Neubot is proceeding in the following directions. We are working to port Neubot on Android [23]. To do so, we are writing an Android application based on a Neubot library partially written in C for performance reasons and partially in a to-be-decided high-level language. We are also working on a new network-performance test for the Micro Transport Protocol (uTP). We are working to improve the plugins architecture, measure the host load, and implement peer-to-peer tests. In parallel, we will also continue the Neuviz effort mentioned above.

Bibliography

[1] Codice in Materia di Protezione dei Dati Personali, 2003. URL http://www.garanteprivacy.it/garante/doc.jsp?ID=1311248.

[2] Glasnost: Results from tests for BitTorrent traffic blocking (20080725), 2008. URL http://broadband.mpi-sws.org/transparency/results/20080725/index.html.

[3] Glasnost: Results from tests for BitTorrent traffic blocking (20081109), 2008. URL http://broadband.mpi-sws.org/transparency/results/20081109/index.html.

[4] IP Addresses Are Personal Data, E.U. Regulator Says, 2008. URL http://www.washingtonpost.com/wp-dyn/content/article/2008/01/21/AR2008012101340.html.

[5] Opinion 1/2008 on Data Protection Issues Related to Search Engines, 2008. URL http://ec.europa.eu/justice/policies/privacy/docs/wpdocs/2008/wp148_en.pdf.

[6] Glasnost: Results from tests for BitTorrent traffic blocking (20090201), 2009. URL http://broadband.mpi-sws.org/transparency/results/20090201/index.html.

[7] NNMA – NNSquad Network Measurement Agent, 2009. URL http://www.nnsquad.org/agent.

[8] Switzerland Network Testing Tool | Electronic Frontier Foundation, 2009. URL https://www.eff.org/pages/switzerland-network-testing-tool.

[9] Peering problems: digging into the Comcast/Level 3 grudgematch, 2010. URL http://arstechnica.com/tech-policy/2010/12/comcastlevel3/.

[10] Netflix Signs Multi-Year Deal with Level 3 for Streaming Services, 2010. URL http://www.businesswire.com/news/home/20101111005421/en/Netflix-Signs-Multi-Year-Deal-Level-3-Streaming.

[11] GeoLite Free Downloadable Databases « Maxmind Developer Site, 2011. URL http://dev.maxmind.com/geoip/legacy/geolite/.

[12] FrontPage - py2exe.org, 2011. URL http://www.py2exe.org/.

[13] Twisted Matrix Labs – Building the engine of your internet, 2011. URL `http://twistedmatrix.com/trac/`.

[14] WindRider – A Mobile Network Neutrality Monitoring System, 2011. URL `http://www.cs.northwestern.edu/~ict992/mobile.htm`.

[15] Youtube Video Speed History, 2011. URL `http://www.youtube.com/my_speed`.

[16] BISmark – Monitor and Manage Your Home Network, 2012. URL `http://projectbismark.net/`.

[17] Net neutrality in slovenia, 2013. URL `https://wlan-si.net/en/blog/2013/06/16/net-neutrality-in-slovenia/`.

[18] M-Lab Visualization: Broadband Performance Using NDT Data, 2014. URL `http://goo.gl/m9WbS`.

[19] SpeedTest.net Visualization: Net Index by Ookla, 2014. URL `http://www.netindex.com/`.

[20] Network Neutrality Map Using Glasnost Data, 2014. URL `http://netneutralitymap.org/`.

[21] The Network is Aware Project, 2014. URL `http://dpi.ischool.syr.edu/Home.html`.

[22] Deep Packet Inspection Stats Using Glasnost Data, 2014. URL `http://dpi.ischool.syr.edu/MLab-Data.html`.

[23] Android Operating System, 2014. URL `http://en.wikipedia.org/wiki/Android_%28operating_system%29`.

[24] Android NDK, 2014. URL `https://developer.android.com/tools/sdk/ndk/index.html`.

[25] A comparison of Apache and Yaws, 2014. URL `http://www.sics.se/~joe/apachevsyaws.html`.

[26] BigQuery - Google Cloud Platform, 2014. URL `https://cloud.google.com/products/bigquery/`.

[27] BitTorrent - Delivering the World's Content, 2014. URL `http://www.bittorrent.com`.

[28] Bandwidth Cap, 2014. URL `http://en.wikipedia.org/wiki/Bandwidth_cap`.

[29] Compatibility of C and C++, 2014. URL `http://en.wikipedia.org/wiki/Compatibility_of_C_and_C%2B%2B`.

[30] Creative Commons - CC0 1.0 Universal, 2014. URL http://creativecommons.org/publicdomain/zero/1.0/.

[31] Congestion Avoidance Overview, 2014. URL http://www.cisco.com/c/en/us/td/docs/ios/12_2/qos/configuration/guide/fqos_c/qcfconav.html.

[32] VNI Forecast Highlights - Cisco Systems, 2014. URL http://www.cisco.com/web/solutions/sp/vni/vni_forecast_highlights/index.html.

[33] Google Cloud Storage: m-lab/glasnost, 2014. URL https://console.developers.google.com/storage/m-lab/glasnost/.

[34] Google Cloud Storage: m-lab, 2014. URL https://cloud.google.com/console/storage/m-lab.

[35] Google Cloud Storage: m-lab/ndt, 2014. URL https://console.developers.google.com/storage/m-lab/ndt/.

[36] Google Cloud Storage: m-lab/neubot, 2014. URL https://cloud.google.com/console/storage/m-lab/neubot/.

[37] Google Cloud Storage: m-lab/npad, 2014. URL https://console.developers.google.com/storage/m-lab/npad/.

[38] Google Cloud Storage: m-lab/shaperprobe, 2014. URL https://console.developers.google.com/storage/m-lab/shaperprobe/.

[39] Analytics for a Digital World - comScore, Inc., 2014. URL http://www.comscore.com/.

[40] D3.js - Data-Driver Documents, 2014. URL http://d3js.org/.

[41] Dalvik, 2014. URL https://code.google.com/p/dalvik/.

[42] MPEG-DASH / Media Source demo, 2014. URL https://dash-mse-test.appspot.com/.

[43] Dasu – A platform for measurement experimentation and broadband characterization, 2014. URL http://www.aqualab.cs.northwestern.edu/projects/115-dasu-isp-characterization-from-the-network-edge.

[44] DiffProbe, 2014. URL http://netinfer.net/diffprobe/.

[45] DNS avert by Hal Pomeranz, 2014. URL http://www.deer-run.com/~hal/sysadmin/dns-advert.html.

[46] Erlang Programming Language, 2014. URL http://www.erlang.org/.

[47] Fathom: a Browser-Based Network Measurement Platform, 2014. URL `http://fathom.icsi.berkeley.edu/`.

[48] android/platform_dalvik, 2014. URL `https://github.com/android/platform_dalvik`.

[49] neubot/neuviz, 2014. URL `https://github.com/neubot/neuviz`.

[50] Glasnost – Google Project Hosting, 2014.

[51] Glasnost: Test if your ISP is shaping your traffic, 2014. URL `http://broadband.mpi-sws.org/transparency/glasnost.php`.

[52] The Go Programming Language, 2014. URL `http://golang.org/`.

[53] Public DNS - Google Developers, 2014. URL `https://developers.google.com/speed/public-dns/`.

[54] Smart disclosure, 2014. URL `http://thegovlab.org/wiki/Smart_disclosure`.

[55] The GNU General Public License v3.0 - GNU Project - Free Software Foundation, 2014. URL `https://www.gnu.org/copyleft/gpl.html`.

[56] Grenouille.com - la météo du net depuis 2000, 2014. URL `http://www.grenouille.com/`.

[57] Association - Grenouille Wiki, 2014. URL `http://wiki.grenouille.com/index.php/Association`.

[58] Grenouille FAQ, 2014. URL `http://wiki.grenouille.com/index.php/Projet_FAQ`.

[59] Grenouille.com Team's Git Repository, 2014. URL `http://git.grenouille.com/`.

[60] Grenouille.com, 2014. URL `http://fr.wikipedia.org/wiki/Grenouille.com`.

[61] Welcome to Apache™ Hadoop®!, 2014. URL `http://hadoop.apache.org/`.

[62] Herdict: Browse Lists, 2014. URL `http://www.herdict.org/lists`.

[63] HoBBIT - Host Based Broadband Internet Telemetry, 2014. URL `http://new.hobbit.comics.unina.it/`.

[64] SamKnows broadband measurement platform and how it works in the home, 2014. URL `https://www.samknows.com/broadband/how-it-works`.

[65] joyent/http-parser, 2014. URL `https://github.com/joyent/http-parser`.

[66] IANA: Service Name and Transport Protocol Port Number Registry, 2014. URL http://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml.

[67] iOS, 2014. URL http://en.wikipedia.org/wiki/IOS.

[68] Net Neutrality II, 2014. URL http://israeltechnologylaw.wordpress.com/2014/02/13/net-neutrality-ii/.

[69] Java Programming Language, 2014. URL http://en.wikipedia.org/wiki/Java_%28programming_language%29.

[70] JSON, 2014. URL http://json.org/.

[71] The Jython Project, 2014. URL http://www.jython.org/.

[72] Launchd: core.c, 2014. URL http://www.opensource.apple.com/source/launchd/launchd-842.1.4/src/core.c.

[73] Lauren Weinstein (technologist), 2014. URL http://en.wikipedia.org/wiki/Lauren_Weinstein_%28technologist%29.

[74] Libdash - bitmovin GmbH, 2014. URL http://www.bitmovin.net/libdash.html.

[75] libevent, 2014. URL http://libevent.org/.

[76] libffi, 2014. URL https://sourceware.org/libffi/.

[77] bittorrent/libutp, 2014. URL https://github.com/bittorrent/libutp.

[78] joyent/libuv, 2014. URL https://github.com/joyent/libuv.

[79] Linux-VServer, 2014. URL http://linux-vserver.org.

[80] The Lua Programming Language, 2014. URL http://www.lua.org/.

[81] The MeasrDroid Project, 2014. URL http://www.droid.net.in.tum.de/about/measurements.

[82] Neubot Data Analysis – Dashboard, 2014. URL http://media.polito.it/neubot.

[83] Neubot Data Analysis – World, 2014. URL http://media.polito.it/neubot/world.

[84] MisuraInternet.it, 2014. URL https://www.misurainternet.it/.

[85] HowToAccessMLabData - m-lab [...], 2014. URL https://code.google.com/p/m-lab/wiki/HowToAccessMLabData.

[86] NewSiteDeploymentProcess - m-lab [...], 2014. URL `https://code.google.com/p/m-lab/wiki/NewSiteDeploymentProcess`.

[87] Mlab_ns source code at code.google.com, 2014. URL `https://code.google.com/p/m-lab`.

[88] MobiPerf, 2014. URL `http://www.mobiperf.com/`.

[89] neubot/mod_dash at 0.4.16.9 - neubot/neubot - Github, 2014. URL `https://github.com/neubot/neubot/tree/0.4.16.9/mod_dash`.

[90] MongoDB, 2014. URL `http://www.mongodb.org/`.

[91] MorFEO | Nexa Center for Internet & Society, 2014. URL `http://nexa.polito.it/morfeo`.

[92] Home of the Mozilla Project – Mozilla, 2014. URL `http://www.mozilla.org`.

[93] MTR, 2014. URL `http://www.bitwizard.nl/mtr/`.

[94] Nagios - The Industry Standard in IT Infrastructure Monitoring, 2014. URL `http://www.nagios.org/`.

[95] GTNOISE NANO Project / code.php, 2014. URL `http://gtnoise.net/nano/code.php`.

[96] NANO | GT Noise, 2014. URL `http://noise-lab.net/projects/old-projects/nano/`.

[97] NDT – Google Project Hosting, 2014.

[98] netem | The Linux Foundation, 2014. URL `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem`.

[99] The Netfilter Project, 2014. URL `http://www.netfilter.org/`.

[100] neubot/doc/faq.md - neubot/neubot - Github, 2014. URL `https://github.com/neubot/neubot/blob/0.4.16.9/doc/faq.md`.

[101] neubot/PRIVACY - neubot/neubot - Github, 2014. URL `https://github.com/neubot/neubot/blob/0.4.16.9/PRIVACY`.

[102] neubot/neubot/backend_mlab.py at 0.4.16.9 - neubot/neubot, 2014. URL `https://github.com/neubot/neubot/blob/0.4.16.9/neubot/backend_mlab.py`.

[103] Neubot data, 2014. URL `http://data.neubot.org/`.

[104] neubot/doc/neubot.1.rst at 0.4.16.9 - neubot/neubot, 2014. URL `https://github.com/neubot/neubot/blob/master/doc/neubot.1.rst#web-api`.

[105] neubot/neubot/updater/unix.py at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/neubot/updater/unix.py.

[106] neubot/neubot/www/js/results.js at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/neubot/www/js/results.js#L41.

[107] neubot/doc/neubot.1.rst at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/doc/neubot.1.rst#data-processing-language.

[108] neubot/neubot/runner_hosts.py at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/neubot/runner_hosts.py#L209.

[109] neubot/neubot/runner_policy.py at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/neubot/runner_policy.py#L37.

[110] neubot/MasterSrv/redir_table.py at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/MasterSrv/redir_table.py.

[111] neubot/neubot/rendezvous/server.py at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/neubot/rendezvous/server.py.

[112] neubot/neubot/rendezvous/server.py at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/neubot/server.py#L137.

[113] neubot/doc/faq.md at 0.4.16.9 - neubot/neubot, 2014. URL https://github.com/neubot/neubot/blob/0.4.16.9/doc/faq.md#7-web-interface.

[114] neubot/neuviz, 2014. URL https://github.com/neubot/neuviz.

[115] comScore ISP Speed Test Accuracy, 2014. URL http://www.netforecast.com/Reports/NFR5103_comScore_ISP_Speed_Test_Accuracy.pdf.

[116] Network Neutrality FAQ, 2014. URL http://www.timwu.org/network_neutrality.html.

[117] Network Neutrality Squad, 2014. URL http://www.nnsquad.org/.

[118] node.js, 2014. URL http://nodejs.org/.

[119] CVS log for src/lib/libc/gen/daemon.c, 2014. URL http://www.openbsd.org/cgi-bin/cvsweb/src/lib/libc/gen/daemon.c.

[120] Ookla | The world standard in Internet metrics, 2014. URL https://www.ookla.com/.

[121] OONI raw reports, 2014. URL https://ooni.torproject.org/reports/.

[122] gitweb.torproject.org – ooni-probe.git/summary, 2014. URL `https://gitweb.torproject.org/ooni-probe.git`.

[123] OpenWrt - wireless freedom, 2014. URL `https://openwrt.org/`.

[124] PF: The OpenBSD Packet Filter, 2014. URL `http://www.openbsd.org/faq/pf/`.

[125] PF: Packet Queueing and Prioritization, 2014. URL `http://www.openbsd.org/faq/pf/queueing.html`.

[126] Portolan – Network Sensing Architecture, 2014. URL `http://portolan.iet.unipi.it/`.

[127] Waiting for I/O completion, 2014. URL `http://docs.python.org/2/library/select.html`.

[128] Low-level networking interface, 2014. URL `http://docs.python.org/2/library/socket.html`.

[129] pygrenouille.git/summary, 2014. URL `http://git.grenouille.com/?p=pygrenouille.git;a=summary`.

[130] Welcome to Python.org, 2014. URL `http://www.python.org/`.

[131] Respect My Net, 2014. URL `http://respectmynet.eu/`.

[132] rsync, 2014. URL `http://rsync.samba.org/`.

[133] Neubot Data Analysis, 2014. URL `http://streaming.polito.it/neubot/`.

[134] ShaperProbe – Google Project Hosting, 2014. URL `https://code.google.com/p/shaperprobe/`.

[135] ShaperProbe: Detecting ISP Traffic Rate-Limiting – Google Project Hosting, 2014. URL `https://code.google.com/p/shaperprobe/source/browse/`.

[136] Speedtest.net - The Global Broadband Speed Test, 2014. URL `http://speedtest.net/`.

[137] Simplified Wrapper and Interface Generator, 2014. URL `http://www.swig.org/`.

[138] TCPDUMP/LIBPCAP public repository, 2014. URL `http://www.tcpdump.org/`.

[139] tcptrace - Official Homepage, 2014. URL `http://www.tcptrace.org/`.

[140] Neubot TestbedKit for W-MUST 2012, 2014. URL `http://data.neubot.org/research/2012-wmust/testbedkit`.

[141] Torino Piemonte Internet Exchange, 2014. URL `http://www.top-ix.org/`.

[142] VideoLAN - Official page for the VLC media player, the Open Source video framework!, 2014. URL http://www.videolan.org/vlc/index.html.

[143] Vodafone UK traffic management for mobile dongles, laptops, netbooks, iPads and tablets, 2014. URL http://www.vodafone.co.uk/cs/groups/public/documents/webcontent/vftst073790.pdf.

[144] Vodafone UK traffic management for mobile phones, 2014. URL http://www.vodafone.co.uk/cs/groups/public/documents/webcontent/vftst073789.pdf.

[145] Terms and conditions - Traffic Management - Vodafone, 2014. URL http://www.vodafone.co.uk/about-this-site/terms-and-conditions/traffic-management/.

[146] Azureus - now called Vuze - Bittorrent Client: the Dasu plugin, 2014. URL http://plugins.vuze.com/plugin_details.php?plugin=dasu.

[147] Actor Model, 2014. URL http://en.wikipedia.org/wiki/Actor_model.

[148] Common Gateway Interface, 2014. URL http://en.wikipedia.org/wiki/Common_Gateway_Interface.

[149] Comet (programming), 2014. URL http://en.wikipedia.org/wiki/Comet_%28programming%29.

[150] Data wrangling, 2014. URL http://en.wikipedia.org/wiki/Data_wrangling.

[151] Differentiated Services, 2014. URL http://en.wikipedia.org/wiki/Differentiated_services.

[152] Domain Name System, 2014. URL http://en.wikipedia.org/wiki/Domain_Name_System.

[153] Google App Engine, 2014. URL http://en.wikipedia.org/wiki/Google_App_Engine.

[154] Iperf, 2014. URL http://en.wikipedia.org/wiki/Iperf.

[155] Java Native Interface, 2014. URL http://en.wikipedia.org/wiki/Java_Native_Interface.

[156] LAMP (software bundle), 2014. URL http://en.wikipedia.org/wiki/LAMP_%28software_bundle%29.

[157] Lisp Programming Language, 2014. URL http://en.wikipedia.org/wiki/Lisp_%28programming_language%29.

[158] Multiprotocol Label Switching, 2014. URL http://en.wikipedia.org/wiki/Multiprotocol_Label_Switching.

[159] Net neutrality, 2014. URL http://en.wikipedia.org/wiki/Net_neutrality.

[160] Onion routing, 2014. URL http://en.wikipedia.org/wiki/Onion_routing.

[161] Peering, 2014. URL http://en.wikipedia.org/wiki/Peering.

[162] Public-Key Infrastructure, 2014. URL http://en.wikipedia.org/wiki/Public-key_infrastructure.

[163] Proxy Server, 2014. URL http://en.wikipedia.org/wiki/Proxy_server.

[164] Reactor Pattern, 2014. URL http://en.wikipedia.org/wiki/Reactor_pattern.

[165] Trusted Computing, 2014. URL http://en.wikipedia.org/wiki/Trusted_Computing.

[166] Virtual Private Network, 2014. URL http://en.wikipedia.org/wiki/VPN.

[167] XML, 2014. URL http://en.wikipedia.org/wiki/XML.

[168] Goodput, 2014. URL http://en.wikipedia.org/wiki/Goodput.

[169] The Wireshark Wiki: BitTorrent, 2014. URL http://wiki.wireshark.org/BitTorrent.

[170] S. Akhshabi, A. C. Begen, and C. Dovrolis. An Experimental Evaluation of Rate-Adaptation Algorithms in Adaptive Streaming over HTTP. In *Multimedia Systems Conference*, pages 157–168. ACM, 2011.

[171] M. Allman, V. Paxson, and E. Blanton. TCP Congestion Control. RFC 5681, RFC Editor, September 2009. URL http://www.rfc-editor.org/rfc/rfc5681.txt.

[172] C. Anderson. Dimming the Internet: Detecting Throttling as a Mechanism of Censorship in Iran. *arXiv preprint arXiv:1306.4361*, 2013.

[173] H. Asghari, M. Van Eeten, and M. Mueller. Unraveling the Economic and Political Drivers of Deep Packet Inspection. In *GigaNet Annual Symposium*, 2012.

[174] M. Baldi, A. Baldini, N. Cascarano, and F. Risso. Service-Based Traffic Classification: Principles and Validation. In *Sarnoff Symposium*, pages 1–6. IEEE, 2009.

[175] S. Basso, A. Servetti, and J. C. De Martin. The Hitchhiker's Guide to the Network Neutrality Bot Test Methodology. In *Congresso Nazionale AICA*. Associazione Italiana per l'Informatica ed il Calcolo Automatico, 2011.

[176] S. Basso, A. Servetti, and J. C. De Martin. The Network Neutrality Bot Architecture: a Preliminary Approach for Self-Monitoring of Internet Access QoS. In *IEEE Symposium on Computers and Communications*, pages 1131–1136. IEEE, 2011.

[177] S. Basso, M. Meo, A. Servetti, and J. C. De Martin. Estimating Packet Loss Rate in the Access through Application-Level Measurements. In *ACM SIGCOMM Workshop on Measurements Up the Stack*, pages 7–12. ACM, 2012.

[178] S. Basso, M. Meo, and J. C. De Martin. Strengthening Measurements from the Edges: Application-Level Packet Loss Rate Estimation. *ACM SIGCOMM Computer Communication Review*, 43(3):45–51, 2013.

[179] S. Basso, A. Servetti, E. Masala, and J. C. De Martin. Measuring DASH Streaming Performance from the End Users Perspective using Neubot. In *Multimedia Systems Conference*. ACM, 2014.

[180] S. Bauer, D. Clark, and W. Lehr. Understanding broadband speed measurements. *MIT Internet Traffic Analysis Study (MITAS) project white paper*, 2010.

[181] S. Bauer, D. Clark, and W. Lehr. Powerboost. In *ACM SIGCOMM Workshop on Home Networks*, pages 7–12. ACM, 2011.

[182] D. M. Beazley. SWIG: An Easy to Use Tool for Integrating Scripting Languages with C And C++. In *Annual USENIX Tcl/Tk Workshop*, pages 129–139. USENIX Association, 1996.

[183] D. J. Bernstein. Non-blocking BSD Socket Connections, 2014. URL `http://cr.yp.to/docs/connect.html`.

[184] R. Beverly, S. Bauer, and A. Berger. The internet is not a big truck: toward quantifying network neutrality. In *Passive and Active Network Measurement*, pages 135–144. Springer, 2007.

[185] Z. S. Bischof, J. S. Otto, and F. E. Bustamante. Up, Down and Around the Stack: ISP Characterization from Network Intensive Applications. *ACM SIGCOMM Computer Communication Review*, 42(4):515–520, 2012.

[186] D. Bonfiglio, M. Mellia, M. Meo, D. Rossi, and P. Tofanelli. Revealing Skype Traffic: when Randomness Plays with You. *ACM SIGCOMM Computer Communication Review*, 37(4):37–48, 2007.

[187] M. Boucadair, R. Penno, and D. Wing. Universal Plug and Play (UPnP) Internet Gateway Device - Port Control Protocol Interworking Function (IGD-PCP IWF). RFC 6970, RFC Editor, July 2013. URL `http://www.rfc-editor.org/rfc/rfc6970.txt`.

[188] J. Brodkin. Why YouTube buffers: The secret deals that make–and break–online video, 2013. URL http://arstechnica.com/information-technology/2013/07/why-youtube-buffers-the-secret-deals-that-make-and-break-online-video/.

[189] E. Brosh, S. A. Baset, V. Misra, D. Rubenstein, and H. Schulzrinne. The Delay-Friendliness of TCP for Real-Time Traffic. *IEEE/ACM Transactions on Networking*, 18(5):1478–1491, 2010.

[190] C. Caini and R. Firrincieli. TCP Hybla: a TCP Enhancement for Heterogeneous Networks. *International Journal of Satellite Communications and Networking*, 22 (5):547–566, 2004.

[191] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *International Conference on Computer Communications (INFOCOM)*, pages 1742–1751. IEEE, 2000.

[192] R. Carlson. Developing the Web100 Based Network Diagnostic Tool (NDT). In *Passive and Active Measurement Workshop*, 2003.

[193] G. Carofiglio, L. Muscariello, D. Rossi, C. Testa, and S. Valenti. Rethinking the Low Extra Delay Background Transport (LEDBAT) Protocol. *Computer Networks*, 57 (8):1838–1852, 2013.

[194] C. Chirichella and D. Rossi. To the Moon and Back: are Internet Bufferbloat Delays Really that Large? In *International Conference on Computer Communications (INFOCOM), Traffic Measurement and Analysis Workshop*. IEEE, 2013.

[195] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. Planetlab: an Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):3–12, 2003.

[196] B. Cohen. The BitTorrent Protocol Specification, 2008. URL http://www.bittorrent.org/beps/bep_0003.html.

[197] B. Cohen. TCP Sucks, 2012. URL http://bramcohen.com/2012/05/07/tcp-sucks.

[198] J. O. Coplien and D. C. Schmidt. *Pattern Languages of Program Design*. ACM Press/Addison-Wesley Publishing Co., 1995.

[199] J. Crowcroft. Net Neutrality: the Technical Side of the Debate: a White Paper. *ACM SIGCOMM Computer Communication Review*, 37(1):49–56, 2007.

[200] W. de Donato, A. Botta, and A. Pescapé. HoBBIT: A Platform for Monitoring Broadband Performance from the User Network. In *Traffic Monitoring and Analysis*, pages 65–77. Springer, 2014.

[201] J. C. De Martin and A. Glorioso. The Neubot Project: A Collaborative Approach to Measuring Internet Neutrality. In *International Symposium on Technology and Society*, pages 1–4. IEEE, 2008.

[202] M. Dhawan, J. Samuel, R. Teixeira, C. Kreibich, M. Allman, N. Weaver, and V. Paxson. Fathom: a Browser-Based Network Measurement Platform. In *Internet Measurement Conference*, pages 73–86. ACM, 2012.

[203] M. Dischinger, A. Haeberlen, K. P. Gummadi, and S. Saroiu. Characterizing Residential Broadband Networks. In *Internet Measurement Conference*, pages 43–56. ACM, 2007.

[204] M. Dischinger, A. Mislove, A. Haeberlen, and K. P. Gummadi. Detecting BitTorrent Blocking. In *Internet Measurement Conference*, pages 3–8. ACM, 2008.

[205] M. Dischinger, M. Marcon, S. Guha, K. P. Gummadi, R. Mahajan, and S. Saroiu. Glasnost: Enabling End Users to Detect Traffic Differentiation. In *Conference on Networked Systems Design and Implementation*, pages 27–27. USENIX Association, 2010.

[206] C. Dovrolis, K. P. Gummadi, A. Kuzmanovic, and S. D. Meinrath. Measurement Lab: Overview and an Invitation to the Research Community. *ACM SIGCOMM Computer Communication Review*, 40(3):53–56, 2010.

[207] P. Eckersley. How Unique is your Web Browser? In *Privacy Enhancing Technologies*, pages 1–18. Springer, 2010.

[208] J. Engebretson. Behind the Level 3 – Comcast Peering Settlement, 2013. URL `http://www.telecompetitor.com/behind-the-level-3-comcast-peering-settlement/`.

[209] G. R. Faulhaber. Transparency and Broadband Internet Service Providers. *International Journal of Communication*, 4:20, 2010.

[210] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455, RFC Editor, December 2011. URL `http://www.rfc-editor.org/rfc/rfc6455.txt`.

[211] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol–HTTP/1.1. RFC 2616, RFC Editor, June 1999. URL `http://www.rfc-editor.org/rfc/rfc2616.txt`.

[212] A. Filastó and J. Appelbaum. Ooni: Open Observatory of Network Interference. In *Workshop on Free and Open Communications on the Internet*. USENIX Association, 2012.

[213] M. Fisk and W.-c. Feng. Dynamic Right-Sizing in TCP. In *Los Alamos Computer Science Institute Symposium*, volume 1, 2001.

[214] S. Floyd. HighSpeed TCP for Large Congestion Windows. RFC 3649, RFC Editor, December 2003. URL http://www.rfc-editor.org/rfc/rfc3649.txt.

[215] B. Ford, P. Srisuresh, and D. Kegel. Peer-to-Peer Communication Across Network Address Translators. In *USENIX Annual Technical Conference*, pages 179–192. USENIX Association, 2005.

[216] B. M. Frischmann. An Economic Theory of Infrastructure and Commons Management. *Minnesota Law Review*, 89:917–1030, 2005.

[217] G. Futia, E. Zimuel, S. Basso, and J. C. De Martin. Visualizing Internet-Measurements Data for Research Purposes: the NeuViz Data Visualization Tool. In *Congresso Nazionale AICA*. Associazione Italiana per l'Informatica ed il Calcolo Automatico, 2013.

[218] G. Futia, E. Zimuel, S. Basso, and J. C. De Martin. The NeuViz Data Visualization Tool for Visualizing Internet-Measurements Data. *Mondo Digitale*, pages 1–16, 2014.

[219] J. Gettys and K. Nichols. Bufferbloat: Dark buffers in the internet. *ACM Queue*, 9 (11):40, 2011.

[220] E. Gregori, L. Lenzini, V. Luconi, and A. Vecchio. Sensing the Internet through crowdsourcing. In *International Conference on Pervasive Computing and Communications Workshops*, pages 248–254. IEEE, 2013.

[221] F. Guillemin, P. Robert, and B. Zwart. Performance of TCP in the Presence of Correlated Packet Loss. In *15th ITC Specialist Seminar on Internet Traffic Engineering and Traffic Management (Wurzburg)*, 2002.

[222] S. Ha, I. Rhee, and L. Xu. CUBIC: a New TCP-Friendly High-Speed TCP Variant. *ACM SIGOPS Operating Systems Review*, 42(5):64–74, 2008.

[223] D. Hamon. Visualizing M-Lab Data with BigQuery, 2012. URL http://dmadev.com/2012/11/19/.

[224] T. Henderson, S. Floyd, A. Gurtov, and Y. Nishida. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 6582, RFC Editor, April 2012. URL http://www.rfc-editor.org/rfc/rfc6582.txt.

[225] J. C. Hoe. Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes. Master's thesis, Massachusetts Institute of Technology, 1995.

[226] T. Hwang. Herdict: a Distributed Model for Threats Online. *Network Security*, 2007 (8):15–18, 2007.

[227] ISO/IEC DIS 23009-1. Information technology – Dynamic adaptive streaming over HTTP (DASH) – Part 1: Media presentation description and segment formats, 2012.

[228] V. Jacobson. Congestion Avoidance and Control. *ACM SIGCOMM Computer Communication Review*, 18(4):314–329, 1988.

[229] A. Joch. Debating Net Neutrality. *Communications of the ACM*, 52(10):14–15, 2009.

[230] P. Kanuparthy and C. Dovrolis. Diffprobe: Detecting ISP Service Discrimination. In *International Conference on Computer Communications (INFOCOM)*, pages 1–9. IEEE, 2010.

[231] P. Kanuparthy and C. Dovrolis. ShaperProbe: End-to-End Detection of ISP Traffic Shaping Using Active Methods. In *Internet Measurement Conference*, pages 473–482. ACM, 2011.

[232] T. Karagiannis, A. Broido, N. Brownlee, K. Claffy, and M. Faloutsos. Is P2P Dying or Just Hiding? In *Global Telecommunications Conference (GLOBECOM)*, pages 1532–1538. IEEE, 2004.

[233] D. Kegel. The C10K Problem, 1999. URL `http://www.kegel.com/c10k.html`.

[234] T. Kelly. Scalable TCP: Improving Performance in Highspeed Wide Area Networks. *ACM SIGCOMM Computer Communication Review*, 33(2):83–91, 2003.

[235] J. Krämer, L. Wiewiorra, and C. Weinhardt. Net neutrality: A progress report. *Telecommunications Policy*, 37(9):794–813, 2013.

[236] D. Kravets. Comcast No Longer Chocking File Sharers' Connections, Study Says, 2011. URL `http://www.wired.com/threatlevel/2011/10/bittorrent-throttling-comcast/`.

[237] C. Kreibich, N. Weaver, B. Nechaev, and V. Paxson. Netalyzr: illuminating the edge network. In *Internet Measurement Conference*, pages 246–259. ACM, 2010.

[238] S. Lederer, C. Müller, and C. Timmerer. Dynamic Adaptive Streaming over HTTP Dataset. In *Multimedia Systems Conference*, pages 89–94. ACM, 2012.

[239] S. Lederer, C. Müller, C. Timmerer, C. Concolato, J. Le Feuvre, and K. Fligegel. Distributed DASH Dataset. In *Multimedia Systems Conference*, pages 131–135. ACM, 2013.

[240] M. Lemley and L. Lessig. End of End-to-End: Preserving the Architecture of the Internet in the Broadband Era. *UCLA Law Review*, 48:925, 2000.

[241] C. Liu, I. Bouazizi, and M. Gabbouj. Rate Adaptation for Adaptive HTTP Streaming. In *Multimedia Systems Conference*, pages 169–174. ACM, 2011.

[242] A. Livingstone. BitTorrent: Shedding no Tiers, 2006. URL `http://news.bbc.co.uk/2/hi/programmes/newsnight/5017542.stm`.

[243] L. Mamakos, K. Lidl, J. Evarts, D. Carrel, D. Simone, and R. Wheeler. A Method for Transmitting PPP Over Ethernet (PPPoE). RFC 2516, RFC Editor, February 1999. URL http://www.rfc-editor.org/rfc/rfc2516.txt.

[244] C. T. Marsden. *Net Neutrality: Towards a Co-Regulatory Solution*. A&C Black, 2010.

[245] E. Masala, A. Servetti, S. Basso, and J. C. De Martin. Challenges and Issues on Collecting and Analyzing Large Volumes of Network Data Measurements. In *New Trends in Databases and Information Systems*, pages 203–212. Springer, 2014.

[246] S. Mascolo and G. Racanelli. Testing TCP Westwood+ over Transatlantic Links at 10 Gigabit/Second Rate. In *Workshop on Protocols for Fast Long-distance Networks (PFLDnet)*, 2005.

[247] N. Mathewson, P. Syverson, and R. Dingledine. Tor: the Second-Generation Onion Router. In *USENIX Security Symposium*. USENIX Association, 2004.

[248] M. Mathis and J. Mahdavi. Forward Acknowledgement: Refining TCP Congestion Control. *ACM SIGCOMM Computer Communication Review*, 26(4):281–291, 1996.

[249] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, RFC Editor, October 1996. URL http://www.rfc-editor.org/rfc/rfc2018.txt.

[250] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM Computer Communication Review*, 27(3):67–82, 1997.

[251] M. Mathis, J. Heffner, and R. Reddy. Web100: Extended TCP Instrumentation for Research, Education and Diagnosis. *ACM SIGCOMM Computer Communication Review*, 33(3):69–79, 2003.

[252] M. Mathis, J. Heffner, P. O'Neil, and P. Siemsen. Pathdiag: Automated TCP Diagnosis. In *Passive and Active Network Measurement*, pages 152–161. Springer, 2008.

[253] M. Mellia and H. Zhang. TCP Model for Short Lived Flows. *IEEE Communications Letters*, 6(2):85–87, 2002.

[254] M. Mellia, M. Meo, L. Muscariello, and D. Rossi. Passive analysis of tcp anomalies. *Computer Networks*, 52(14):2663–2676, 2008.

[255] A. Mohammed. Verizon Executive Calls for End to Google's 'Free Lunch', 2006. URL http://www.washingtonpost.com/wp-dyn/content/article/2006/02/06/AR2006020601624.html.

[256] A. Moniruzzaman and S. A. Hossain. NoSQL Database: New Era of Databases for Big data Analytics-Classification, Characteristics and Comparison. *International Journal of Database Theory & Application*, 6(4), 2013.

[257] C. Müller, S. Lederer, and C. Timmerer. An Evaluation of Dynamic Adaptive Streaming over HTTP in Vehicular Environments. In *Multimedia Systems Conference: Workshop on Mobile Video*, pages 37–42. ACM, 2012.

[258] B. Nishi. Easier Setup in uTorrent 2.0 and Measurement Lab Collaboration. URL `http://blog.bittorrent.com/2010/01/25/easier-setup-in-%C2%B5torrent-2-0-and-measurement-lab-collaboration/`.

[259] A. Norberg. uTorrent Transport Protocol, 2009. URL `http://www.bittorrent.org/beps/bep_0029.html`.

[260] P. Ohm. Broken Promises of Privacy: Responding to the Surprising Failure of Anonymization. *UCLA Law Review*, 57(6), 2010.

[261] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose. Modeling TCP Throughput: A Simple Model and its Empirical Validation. *ACM SIGCOMM Computer Communication Review*, 28:303–314, 1998.

[262] J. Palfrey and J. Zittrain. Better Data for a Better Internet. *Science*, 334(6060): 1210–1211, 2011.

[263] S. Poojary and V. Sharma. Analytical Model for Congestion Control and Throughput with TCP CUBIC Connections. In *Global Telecommunications Conference (GLOBECOM)*, pages 1–6. IEEE, 2011.

[264] J. Postel. Transmission Control Protocol. RFC 793, RFC Editor, September 1981. URL `http://www.rfc-editor.org/rfc/rfc793.txt`.

[265] R. Prasad, C. Dovrolis, M. Murray, and K. Claffy. Bandwidth Estimation: Metrics, Measurement Techniques, and Tools. *IEEE Network*, 17(6):27–35, 2003.

[266] N. Provos, M. Friedl, and P. Honeyman. Preventing Privilege Escalation. In *USENIX Security Symposium*, pages 231–242. Washington DC, USA, 2003.

[267] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, RFC Editor, September 2001. URL `http://www.rfc-editor.org/rfc/rfc3168.txt`.

[268] C. Reis, A. Barth, and C. Pizano. Browser Security: Lessons from Google Chrome. *ACM Queue*, 7(5):3, 2009.

[269] E. Rescorla. HTTP Over TLS. RFC 2818, RFC Editor, May 2000. URL `http://www.rfc-editor.org/rfc/rfc2818.txt`.

[270] F. Risso, M. Baldi, O. Morandi, A. Baldini, and P. Monclus. Lightweight, Payload-Based Traffic Classification: An Experimental Evaluation. In *International Conference on Communications (ICC)*. IEEE, 2008.

[271] D. Ros and M. Welzl. Assessing LEDBAT's Delay Impact. *Communications Letters, IEEE*, 17(5):1044–1047, 2013.

[272] J. Rosenberg, R. Mahy, P. Matthews, and D. Wing. Session Traversal Utilities for NAT (STUN). RFC 5389, RFC Editor, October 2008. URL `http://www.rfc-editor.org/rfc/rfc5389.txt`.

[273] D. Rossi and S. Valenti. Fine-Grained Traffic Classification with Netflow Data. In *International Wireless Communications and Mobile Computing Conference*, pages 479–483. ACM, 2010.

[274] A. Rothschild. Peering Disputes: Comcast, Level 3, and You, 2010. URL `http://www.internap.com/2010/12/02/peering-disputes-comcast-level-3-and-you/`.

[275] S. Ruehrup, P. Urbano, A. Berger, and A. D'Alconzo. Botnet Detection Revisited: Theory and Practice of Finding Malicious P2P Networks via Internet Connection Graphs. In *International Conference on Computer Communications (INFOCOM)*, pages 3393–3398. IEEE, 2013.

[276] P. Saint-Andre. Extensible Messaging and Presence Protocol (XMPP): Core. RFC 6120, RFC Editor, March 2011. URL `http://www.rfc-editor.org/rfc/rfc6120.txt`.

[277] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-End Arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, 1984.

[278] M. A. Sánchez, J. S. Otto, Z. S. Bischof, D. R. Choffnes, F. E. Bustamante, B. Krishnamurthy, and W. Willinger. Dasu: Pushing Experiments to the Internet's Edge. In *Conference on Networked Systems Design and Implementation*. USENIX Association, 2013.

[279] P. Sarolahti and A. Kuznetsov. Congestion Control in Linux TCP. In *USENIX Annual Technical Conference*, pages 49–62. USENIX Association, 2002.

[280] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550, RFC Editor, July 2003. URL `http://www.rfc-editor.org/rfc/rfc3550.txt`.

[281] L. L. Selwyn. Net Neutrality: A Market Structure Perspective, 2010. URL `http://www.econtech.com/newsletter/june2010/june2010a1.php`.

[282] J. Semke, J. Mahdavi, and M. Mathis. Automatic TCP Buffer Tuning. *ACM SIGCOMM Computer Communication Review*, 28(4):315–323, 1998.

[283] S. Shalunov, G. Hazel, J. Iyengar, and M. Kuehlewind. Low Extra Delay Background Transport (LEDBAT). RFC 6817, RFC Editor, December 2012. URL `http://www.rfc-editor.org/rfc/rfc6817.txt`.

[284] J. P. Sluijs, F. Schuett, and B. Henze. Transparency Regulation as a Remedy for Network Neutrality Concerns: Experimental Results. 2010.

[285] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022, RFC Editor, January 2001. URL `http://www.rfc-editor.org/rfc/rfc3022.txt`.

[286] W. R. Stevens. *UNIX Network Programming*, volume 1. Addison-Wesley Professional, 2004.

[287] S. Sundaresan, W. De Donato, N. Feamster, R. Teixeira, S. Crawford, and A. Pescapè. Broadband Internet Performance: a View from the Gateway. *ACM SIGCOMM Computer Communication Review*, 41(4):134–145, 2011.

[288] S. Sundaresan, N. Feamster, R. Teixeira, A. Tang, W. K. Edwards, R. E. Grinter, M. Chetty, and W. De Donato. Helping Users Shop for ISPs with Internet Nutrition Labels. In *ACM SIGCOMM Workshop on Home Networks*, pages 13–18. ACM, 2011.

[289] B. Swanson. The Coming Exaflood, 2007. URL `http://online.wsj.com/article/SB116925820512582318.html`.

[290] B. Swanson. The Coming Exaflood, 2007. URL `http://www.discovery.org/a/3869`.

[291] K. Tan, J. Song, Q. Zhang, and M. Sridharan. A Compound TCP Approach for High-Speed and Long Distance Networks. Technical report, 2006.

[292] M. Tariq, M. Motiwala, and N. Feamster. NANO: Network Access Neutrality Observatory. In *Workshop on Hot Topics in Networks (HotNets)*. ACM, 2008.

[293] M. Tariq, M. Motiwala, N. Feamster, and M. Ammar. Detecting Network Neutrality Violations with Causal Inference. In *International Conference on Emerging Networking Experiments and Technologies*, pages 289–300. ACM, 2009.

[294] T. C. Thang, A. T. Pham, H. X. Nguyen, P. L. Cuong, and J. W. Kang. Video streaming over HTTP with dynamic resource prediction. In *International Conference on Communications and Electronics*, pages 130–135. IEEE, 2012.

[295] B. Tierney. TCP Tuning Guide for Distributed Application on Wide Area Networks. *USENIX & SAGE Login*, 26(1):33–39, 2001.

[296] A. Tirumala, L. Cottrell, and T. Dunigan. Measuring end-to-end bandwidth with iperf using web100. In *Passive and Active Measurement Workshop*, 2003.

[297] B. Trammell, P. Casas, D. Rossi, Z. Houidi, I. Leontiadis, T. Szemethy, M. Mellia, et al. mplane: an intelligent measurement plane for the internet. *Communications Magazine*, 52(5):148–156, 2014.

[298] D. O. van Daalen (Bits of Freedom). Translations of Key Dutch Internet Freedom Provisions, 2011. URL `https://www.bof.nl/2011/06/27/translations-of-key-dutch-internet-freedom-provisions/`.

[299] B. Van Schewick. Towards an Economic Framework for Network Neutrality Regulation. *Journal on Telecommunications & High Technology Law*, 5:329, 2006.

[300] B. Van Schewick. *Internet architecture and innovation*. MIT Press, 2010.

[301] F. von Lohmann. FCC Rules Against Comcast for BitTorrent Blocking, 2007. URL `https://www.eff.org/deeplinks/2008/08/fcc-rules-against-comcast-bit-torrent-blocking`.

[302] N. Weaver, R. Sommer, and V. Paxson. Detecting Forged TCP Reset Packets. In *Distributed Systems Security Symposium (NDSS)*. Internet Society, 2009.

[303] N. Weil. MPEG-DASH is now industry essential, 2013. URL `https://blogs.akamai.com/2013/11/mpeg-dash-is-now-industry-essential.html`.

[304] T. Wu. Network Neutrality, Broadband Discrimination. *Journal on Telecommunications & High Technology Law*, 2:141, 2003.

[305] Y. Zhang, Z. Mao, and M. Zhang. Ascertaining the Reality of Network Neutrality Violation in Backbone ISPs. In *Workshop on Hot Topics in Networks (HotNets)*, 2008.

[306] Y. Zhang, Z. Mao, and M. Zhang. Detecting Traffic Differentiation in Backbone ISPs with NetPolice. In *Internet Measurement Conference (IMC)*, pages 103–115. ACM, 2009.

[307] J. Zittrain. The Generative Internet. *Harvard Law Review*, pages 1974–2040, 2006.