Doctoral Dissertation
Doctoral Program in Electronics Engineering (29^{th}cycle)

Machine Learning and Big Data Methodologies for Network Traffic Monitoring

By

Danilo Giordano
******

Supervisor:
Prof. Marco Mellia

Politecnico di Torino
2017

Declaration

Acknowledgements

Finally, the last but by no means, least section to write after these intense three years. There are so many people I would like to acknowledge and thank for their support over the past few years, however I will try too keep it short.

Firstly, I would really like to give a huge thank you to my parents and my brother for their help and support throughout this journey, allowing me to make all the choices that has led me to where I am today. I wish to thank Lucrezia and Luca who being apart of my family has helped me through so many tough days. I wish to thank my friend Erica, although being so far away, nonetheless has always been able to maintain a real closeness and I wish thank my friend Kevin for his precious help reviewing this thesis.

I would like to give a general thank you to all my friends and everyone who have been there to support and inspire me throughout my education, thank you very much, it was all truly appreciated.

And finally, I wish to truly thanks my advisor Marco Mellia, who kept pushing me during my PhD career, with his passion for this work and immense patience. So thanks you very much everyone.

Abstract

Over the past 20 years, the Internet saw an exponential grown of traffic, users, services and applications. Currently, it is estimated that the Internet is used everyday by more than 3.6 billions users, who generate 20 TB of traffic per second. Such a huge amount of data challenge network managers and analysts to understand how the network is performing, how users are accessing resources, how to properly control and manage the infrastructure, and how to detect possible threats. Along with mathematical, statistical, and set theory methodologies machine learning and big data approaches have emerged to build systems that aim at automatically extracting information from the raw data that the network monitoring infrastructures offer.

In this thesis I will address different network monitoring solutions, evaluating several methodologies and scenarios. I will show how following a common workflow, it is possible to exploit mathematical, statistical, set theory, and machine learning methodologies to extract meaningful information from the raw data. Particular attention will be given to machine learning and big data methodologies such as DBSCAN, and the Apache Spark big data framework.

The results show that despite being able to take advantage of mathematical, statistical, and set theory tools to characterize a problem, machine learning methodologies are very useful to discover hidden information about the raw data. Using DBSCAN clustering algorithm, I will show how to use *YouLighter*, an unsupervised methodology to group *caches* serving YouTube traffic into *edge-nodes*, and latter by using the notion of *Pattern Dissimilarity*, how to identify changes in their usage over time. By using *YouLighter* over 10-month long traces, I will pinpoint sudden changes in the YouTube *edge-nodes* usage, changes that also impair the end users' Quality of Experience. I will also apply DBSCAN in the deployment of SeLINA, a self-tuning tool implemented in the Apache Spark big data framework to autonomously extract knowledge from network traffic measurements. By using SeLINA, I will show how to automatically detect the changes of the YouTube CDN previously highlighted by *YouLighter*.

Along with these machine learning studies, I will show how to use mathematical and set theory methodologies to investigate the browsing habits of Internauts. By using a two weeks dataset, I will show how over this period, the Internauts continue discovering new websites. Moreover, I will show that by using only DNS information to build a profile, it is hard to build a reliable profiler. Instead, by exploiting mathematical and statistical tools, I will show how to characterize Anycast-enabled CDNs (A-CDNs). I will show that A-CDNs are widely used either for stateless and stateful services. That A-CDNs are quite popular, as, more than 50% of web users contact an A-CDN every day. And that, stateful services, can benefit of A-CDNs, since their paths are very stable over time, as demonstrated by the presence of only a few anomalies in their Round Trip Time.

Finally, I will conclude by showing how I used BGPStream an open-source software framework for the analysis of both historical and real-time Border Gateway Protocol (BGP) measurement data. By using BGPStream in real-time mode I will show how I detected a Multiple Origin AS (MOAS) event, and how I studies the black-holing community propagation, showing the effect of this community in the network. Then, by using BGPStream in historical mode, and the Apache Spark big data framework over 16 years of data, I will show different results such as the continuous growth of IPv4 prefixes, and the growth of MOAS events over time.

All these studies have the aim of showing how monitoring is a fundamental task in different scenarios. In particular, highlighting the importance of machine learning and of big data methodologies.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every day we spend a lot of effort to monitor different aspects of our lives. We monitor the weather, the urban traffic and we use cameras to monitor different areas. Thanks to our smartphones and smartwatches we track our daily activities, how much we walk, how much we sleep, and many other every day activities. Furthermore, the rise of the Internet of Things (IoT) and sensors increases continuously our monitoring capabilities.

In the networking environment, monitoring, has always played a fundamental role in understanding how the network is performing, how users are accessing resources, and how to properly control and manage the infrastructure. The growth of the internet which increased from 100 GB per second and 587 millions users in 2002, to the current 20 TB per second [1] and 3.6 billions users [2], presents an everyday challenge to network analysts, making monitoring a harder task.

On the one hand, this rapid evolution increases the complication of the network, making frequent studies fundamental in keeping the acquired knowledge updated. On the other hand, the growth in term of traffic and available data, calls for the application of novel methodologies, and tools to analyze the network data.

For these reasons in literature, most of the researches target specific problems, e.g., monitoring of a CDNs [3, 4], detecting anomalies [5, 6], profiling behaviours [7]. Although they have different goals, all of them follow a common workflow.

First, data analysts needs to collect network data through *active* or *passive* monitoring. Active monitoring refers to the injection of test traffic into a network to monitor the flow of the traffic. Passive monitoring is an observational study done by placing probes or sniffer in the network to monitor the traffic already present in it. Since these two solutions offer different views of the network, they can be used in synergy to take advantage of both solutions as we will see in Cap. 4 and Cap. 6. Then, after the data acquisition, network analysts have to perform a *feature selection* phase to extract relevant information (*features*) for the addressed problem. It is possible to perform this phase either manually, by having the required domain knowledge, or by using techniques such as the Minimum redundancy-maximum-relevance (mRMR) [9].

After these two steps the data are ready to be analyzed and the skill of the data analysts take in place to extract meaningful information out of them. Along with mathematical and statistical solutions [10, 11], machine learning and big data approaches have emerged to build systems able to extract meaning information from the raw data. Born from pattern recognition, machine learning refers to all the approaches that give computers the ability to learn autonomously. These approaches, which overcome strictly static programs for their ability to predict and make decisions based on the data input, were used for the first time in 1952 by Arthur Samuel for a game of checkers. After that, machine learning algorithms have been widely adopted in different fields such as business [12], medicine [13], or computer science [14, 15], as well as in many other fields. Their extensive application is due to their capabilities to automatically extract information from the data. In networking, machine learning techniques have been used with different goals such as profiling [16], anomaly detection [17, 18], intrusion detection system [19], pattern discovery [20] or monitoring [4].

In addition to the adoption of new methodologies, the exponential growth of the internet creates the necessity of scalable algorithms and tools to process a lot of data in a short time. Big data frameworks address these issues facing three different aspects: (i) the volume of the data as there are a lot of data, (ii), the velocity as data may arrive very quickly and have to be processed fast e.g., streaming application, (iii) the variety of data, as nowadays data, are very heterogeneous e.g., email, videos, audio.

Big data frameworks like hadoop [21] and Spark [22] emerged to solve these problems. Their ability to process a huge amount of data in a short time is due to their distributed architecture, and the possibility to speed-up the process by increasing the available hardware resources e.g., the number of servers (nodes) of the cluster of computers. To take advantage of a distributed architecture, these big data frameworks apply a *MapReduce* programming model. This model created by Google is divided in two phases, first a *map*, and then a *reduce* phase. The map phase is executed in each note separately and it is in charge of reading data from a distributed file system e.g., hadoop file system (hdfs) and perform operation such as *filtering* or *grouping*. The result is that each node loads only part of the data and perform the operations only on that part. After that, the reduce phase groups all the results from all nodes to compute the final result. The key aspect of this model is that by dividing the dataset, each node needs to perform the operations only on part of it. Making the parallel computation extremely fast. However, this programming model works only if it is possible to perform the same operation on each part of the dataset autonomously. These aspects are playing a key role to perform monitoring analysis that would be unsuitable without them.

In this thesis I will provide a broad study of different monitoring solutions in different scenarios and about some anomaly detection techniques. The goal of the thesis is to show how to characterize the data using solutions from the mathematical and statistical world, how to exploit different machine learning solutions to analyze the data and to perform anomaly detection, and finally to show how a big data framework nowadays represents almost a mandatory skill for any data analyst. For each scenario I will address different problems such as: I) How to retrieve the network data II) Which features represent the problem more accurately III) Which methodology needs to be adopted to solve the problem.

This thesis is structured as follows: Cap. 2 describes *YouLighter*, an unsupervised methodology to unveil changes in the YouTube CDN infrastructure. *YouLighter* is based on DBSCAN a well known clustering algorithm which group *caches* that appear colocated and identical into *edge-nodes*. This automatically unveils the YouTube edge-nodes used by the ISP customers. Next *YouLighter* compares the clustering results obtained from two different time snapshots to pinpoint sudden changes exploiting a new metric called *Pattern Dissimilarity*. Cap. 3 investigates the browsing habits of Internauts. By looking the DNS traffic I first present a characterization about how Internauts visit the web. Then, I evaluated if by using the DNS traffic it would be possible to build users' profiles to be used to identify the same user in different time windows. Cap. 4 characterizes the pervasiveness of Anycast-enabled CDNs (A-CDN). First, I show how A-CDNs are used on the internet, giving some insight into eight networks of interest. Then I will present two examples of anomalous change in the RTT behaviour suggesting a possible change in routing information for two A-CDNs. Cap. 5 presents SeLINA a self-tuning tool running on Apache Spark big data framework, to extract knowledge from network traffic measurements. By combining both unsupervised and supervised approaches SeLINA mine data with a scalable approach to check if the new data fits the previous model, to detect possible changes in the traffic, and to, possibly automatically, trigger model rebuilding. Cap. 6 presents BGPStream an open-source software framework for the analysis of both historical and real-time Border Gateway Protocol (BGP) measurement data. By exploiting BGPStream capabilities in different scenarios, I will show how to analyze BGP information through a traditional approach, and how to exploit BGPSteram and Spark big data framework to perform an historical analysis over 16 years of data. In cap. 7 I will draw some considerations and conclusions.

1.1 Datasets

In most of the next chapters I will rely on datasets obtained from passive traces collected from different Points-of-Presence (PoP) of an operational network in an European country-wide ISP. As such, although the dataset will monitor thousands of customers from different areas, they will be bounded in a national area characterized by some geographically and culturally characteristics. Therefore, although the huge amount of clients should represents an heterogeneous scenario, some of my future findings might be biased by the end-users' culturally similarity.

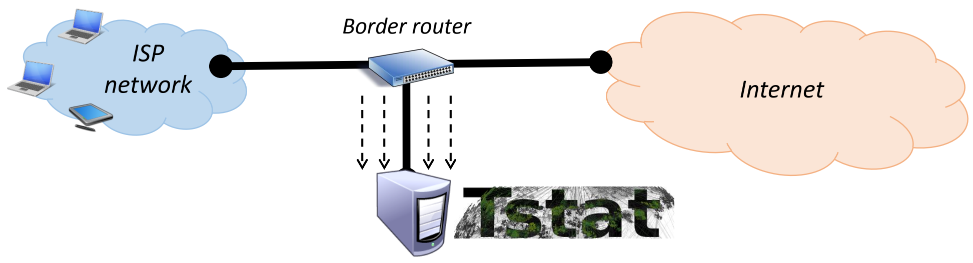


Fig. 1.1 Tstat probe to monitor an ISP PoP

To collect the passive traces, I instrumented several border routers of an European country-wide ISP with a passive probe as depicted in Figure 1.1.

The probe runs Tstat [23] [1], a passive monitoring tool that observes packets flowing on the links connecting the PoP to the ISP backbone network. The probe uses professional Enhance cards to guarantee all packets are efficiently exposed in user-space for processing and able to perform live traffic monitoring up to few Gb/s using off-the-shelf hardware [24, 25]. No sampling is introduced, and the probe is able to process all packets [26]. Tstat rebuilds in real time each TCP and UDP flow, tracks it, and, when the flow is torn down or after an idle timer, it logs more than 100 statistics in a simple text file such as the client IP address, the server IP addresses, the application (L7) protocol type, the amount of bytes and packets sent and received, the TCP Round Trip Time (RTT), The Time To Live (TTL), etc..

To remove any privacy sensitive information from the logs, Tstat anonymizes the client IP address by using irreversible hashing functions, and only aggregate information is considered. The deployment and the information collected for this have been approved by the ISP security and ethic boards. Since Tstat is installed in the middle of the network, in order to monitor the RTT Tstat measures the RTT as the time difference between the server acknowledgement segment and the corresponding client data segment. By evaluating this piece of information Tstat collects all RTT samples and it computes the minimum, the average, and the maximum RTT as the minimum, the average, and the maximum of all samples. Instead The Time To Live (TTL) is directly extracted from the IP header and Tstat compute the same value as the RTT.

Note that Tstat can compute all these metrics considering only TCP segments, and do not require access to application payload. On the one hand, this avoids any privacy issues. And other hand, it allows us to collect all needed statistics even in presence of encryption, e.g. when HTTPS is used.

Tstat implements also DN-Hunter [27], a plugin that annotates each TCP flow with the server Fully Qualified Domain Name (FQDN) the client resolved via previous DNS queries. In more details, when a browser is used to access a website, it has first to perform DNS resolutions of the hostnames associated to the objects found in the page. For each hostname, this returns the IP address(es) of server(s) to contact to fetch the desired object. DN-Hunter parses DNS requests and responses, allowing Tstat to annotate the subsequent TCP flow originated by the same client, and directed to a returned IP address of a server with the original hostname of the service being

[1]http://www.tstat.polito.it

contacted. Since DNS messages are not encrypted, this allows Tstat to extract the hostname for both HTTP and HTTPS traffic with a very limited complexity, thus offering visibility on Web traffic even in presence of encryption.

For instance, assume a client would like to access to *www.acme.com*. It first resolves the hostname into IP address(es) via DNS, getting 123.1.2.3. DN-Hunter caches this information. Then, when later the same client opens a TCP connections to 123.1.2.3, DN-Hunter returns *www.acme.com* from its cache and associate it to the flow. Client DNS cache is rebuild in Tstat, resulting in more than 95% accuracy [27]. With the help of DN-Hunter, it is possible to unveil the hostname being offered by the server having IP address 123.1.2.3, even in presence of encrypted (e.g., HTTPS) or proprietary protocols.

Chapter 2

YouLighter: A Cognitive Approach to Unveil YouTube CDN and Changes

2.1 Introduction

This chapter presents *YouLighter: A Cognitive Approach to Unveil YouTube CDN and Changes* a paper published in the journal *IEEE Transactions on Cognitive Communications and Networking*.[1]

YouTube is one of the most popular and demanding Internet services. It accounts for 1 billion users distributed world-wide, who watch 6 billion hours of videos per month.[2] Due to its popularity and the nature of the content that it distributes, the demanded load to handle is huge, and guaranteeing a satisfactory Quality of Experience (QoE) for the users is a challenging task to accomplish. To this end, YouTube leverages a massive, globally distributed Content Delivery Network (CDN), the Google CDN [28]. It consists of hundreds of *edge-nodes* scattered in the Internet. Each edge-node hosts hundreds of video servers, or *caches*, which can each potentially serve any video any user may request [29].

Google, as many other Over-the-Top content providers, places its edge-nodes close to users, usually at aggregation points directly peering with Internet Service

[1]Danilo Giordano, Stefano Traverso, Luigi Grimaudo, Marco Mellia, Elena Baralis, Alok Tongaonkar, and Sabyasachi Saha "YouLighter: A Cognitive Approach to Unveil YouTube CDN and Changes" in IEEE Transactions on Cognitive Communications and Networking, vol. 1, no. 2, pp. 161-174, June 2015. doi: 10.1109/TCCN.2016.2517004

[2]https://www.youtube.com/yt/press/statistics.html

Providers (ISPs).[3] Hence, Google uses the ISP's network as the "last mile" to deliver YouTube videos. Despite this localized setup, once a user requests a video playback, the CDN load balancing algorithm directs the request to one of the caches, and there is no mean to predict which cache, or even which edge-node will be used [30, 31]. This is particularly critical for the ISP, which on the one hand is compelled to deliver YouTube videos to the customers without impairing the QoE, while on the other aims at minimizing the delivery costs. Hence, the ISP spends a significant effort in monitoring the CDN infrastructure and designing ad hoc traffic engineering policies for YouTube traffic [32]. However, the YouTube CDN allocation policy frequently changes caches being used to serve videos, and changes may involve modifications in the infrastructure, e.g., the activation of a new cache, or in the load balancing algorithm decision, e.g., a sudden switch of caches to serve requests, or an eventual change on the path to those caches, e.g., due to congestion, or route change. Conversely, the ISP's policies are often static and hardly cope with the continuous evolution of the Google CDN: any sudden change can make the ISP's optimization obsolete, and thus ineffective, possibly causing abrupt disruptions or QoE degradations. This constitutes an issue for the ISP, as it sees its reputation degrade when a change happens, even if Google caused it.

2.1.1 My Contribution

In this chapter, I present *YouLighter*, a novel methodology to automatically monitor and pinpoint changes in how the YouTube CDN serves traffic. *YouLighter* relies on an unsupervised learning approach that, as such, does not require any knowledge of the YouTube infrastructure. It builds upon a cognitive approach, where automatic and unsupervised algorithms are used to extract a model of the system status. *YouLighter* only assumes that the ISP has deployed passive traffic probes, which expose TCP flow level logs summarizing video requests from users. Considering a given observation window of, say one day, *YouLighter* aggregates these flow logs to constitute a *snapshot* of the traffic exchanged with YouTube caches. Based on DBSCAN [33], a well-established unsupervised machine learning algorithm, *YouLighter* is able to automatically group thousands of caches into a bunch of edge-nodes using simple features that characterize the network distance of caches from the vantage point.

[3] https://peering.google.com/about/index.html

YouLighter periodically runs DBSCAN on consecutive snapshots, extracting for each of them a model of the status of the CDN. The problem becomes then how to compare the two models to highlight eventual changes. I solve it with some ingenuity, I summarize the corresponding models into *patterns*, and compare them using the notion of *Pattern Dissimilarity*, a metric similar to others presented in the literature, but which satisfies my specific requirements (see Sec. 2.2 for a detailed discussion). The bigger the distance between two snapshots is, the more different the sets of YouTube caches to serve ISP customers during the two periods of time are.

YouLighter highlights several kinds of changes, including deviations from the typical behavior of edge-nodes possibly induced by congestion arising in the network. In general, *YouLighter* triggers alarms corresponding to sudden changes happening in the YouTube CDN infrastructure which may be responsible of QoE issues for ISP customers. Resulting alarms are then offered to the ISP network administrator who can take countermeasures to mitigate the problem.

I validate my methodology over traces I collect from four different vantage points that I have deployed in two ISPs in two different countries. First, I demonstrate that the cognitive algorithms *YouLighter* adopts are effective at identifying and grouping YouTube caches belonging to different edge-nodes. Second, I run *YouLighter* over different collected snapshots considering a longitudinal dataset, which, overall, accounts for more than 33 months of traffic. I pinpoint several examples of sudden and previously undiscovered changes in the YouTube CDN. For some of them, I investigate the impact on the QoE of ISP customers, revealing the sudden drop of average video download throughput to less than 250 kb/s, which hampers even the possibility of watching a video.

I believe that *YouLighter* is a promising tool for ISPs, network administrators and researchers to monitor the YouTube CDN and the traffic it generates. Importantly, thanks to its design, *YouLighter* offers the capability of automating and accelerating the troubleshooting procedures. In fact, ISPs may use *YouLighter* to quickly react to changes possibly harming customer's QoE. For instance, ISPs may adopt traffic engineering algorithms to optimize routing to under-performing edge-nodes, e.g., by means of BGP policies, or to implement DNS policies overruling YouTube choices and re-directing traffic from caches with bad QoE to changes with a good QoE. However, *YouLighter*'s task is limited to notifying the occurrence of change events in the YouTube CDN. The investigation and troubleshooting of issues notified by *YouLighter* are out of the scope of this chapter.

The remainder of this chapter is structured as follows: Sec. 2.2 discusses the related work. Sec. 2.3 describes the details of my datasets, and shows the dynamicity of YouTube cache selection policies. Sec. 2.4 presents my methodology, introduces the notion of *Pattern Dissimilarity*, and discusses *YouLighter*'s complexity. Sec. 2.5 presents my results: First, I evaluate the sensitivity of *YouLighter*'s parameters, and, second, I show how effective *YouLighter* is at pinpointing changes in YouTube CDN employing my traces. Sec. 2.7 suggests some countermeasures an ISP can use to improve users' QoE in case of changes. Finally, Sec. 2.8 concludes the chapter.

2.2 Related Work

A large body of work has analyzed the YouTube delivery infrastructure and its evolution over time [28–30, 34, 32]. They show YouTube is a highly dynamic system which keeps changing over time due to continuous upgrades in the infrastructure [28, 29] or due to the dynamicity of the cache selection policies [30, 34]. Some of the findings are already outdated. For instance, the load-balancing policy based on HTTP redirections which is described in [30, 34] is no longer in place, and YouTube dismissed the naming scheme described in [29] at the end of 2011. In this work, I do not aim to offer an updated view or characterization of YouTube. Instead, I present a methodology that allows to automatically identify changes in both the infrastructure, e.g., the appearance of new edge-nodes, and in the day to day management of the infrastructure, e.g., a change in the load-balancing algorithm that may affect millions of customers.

My contribution is in line with the body of works focusing on anomaly detection, for which [35, 36] offer good surveys. In particular, my work belongs to the family of studies which addresses the problem of performing anomaly detection in large scale operational networks. [37, 38] are notable examples of supervised methodologies which leverage data from passive probes, topology information, routing tables and Simple Network Management Protocol (SNMP) logs to match predictions to actual measurements to pinpoint deviations.

Other works propose methodologies to perform anomaly detection on CDN infrastructures for video delivery specifically [39, 40, 32]. Authors of [39] consider a collection of video download sessions, out of which they extract measurements for specific features, and manually set thresholds to label degradation-affected sessions. Then, by applying graph techniques, they identify clusters and outliers possibly associated to performance issues. [40] analyzes different CDN providers, among which YouTube. The paper presents a characterization of the YouTube cache selection policy and apply an anomaly detection system based on subnet usage to detect unexpected cache selection in a time window of minutes. Finally, [32] focuses on the YouTube case too and proposes a methodology for anomaly detection which requires a significant manual effort, and the paper mostly presents results about the characterization of the YouTube service in terms of traffic characteristics and QoE perceived by the users. *YouLighter* differs from the supervised methodologies described in above studies. In fact, *YouLighter* does not assume any knowledge of a baseline, and leverages unsupervised algorithms to automatically unveil changes on the YouTube infrastructure. I design it with this specific requirement in mind, as it has to target the YouTube CDN, for which the ground truth is a moving target that is very difficult to know.

The application of cognitive and unsupervised machine learning techniques – in particular clustering techniques – to perform anomaly detection based on network traffic is not new. For instance, [17] proposes a flow-based anomaly detection algorithm based on K-Means, while [18] uses DBSCAN to group P2P sessions and identify anomalous clusters. However, in all the cases, clustering is used to study the same given dataset. My goal is anomaly detection algorithm builds on the comparison between clustering patterns obtained at different times. The task of identifying anomalies by comparing clustering results attained from different datasets (e.g., different time snapshots, etc.) translates in the problem of measuring the dissimilarity of two distinct patterns. The comparison and quantification of the similarity obtained from evolving-in-time clustering is addressed within a sub-domain referred to as *online clustering* or *streaming clustering*. See [41, 42] for notable examples. In particular, [41] defines a metric to quantify the dissimilarity between consecutive clustering patterns, namely the History Cost, which is similar to the Pattern Dissimilarity presented in this work. However, the History Cost is designed to compute the distance between clustering patterns built by K-Means, for which the number of clusters is constant across different patterns. Instead, the

Pattern Dissimilarity takes into account the case in which the number of clusters across different patterns is different. Another work which goes in a similar direction is [43] whose authors propose to measure similarity between sets of overlapping clusters from complex networks, in which groups of nodes form tightly connected units linked to each other. Since points are not embedded in a metric space, authors of [43] define ad-hoc distances.

YouLighter differs also from techniques for the tracking of moving clusters and objects as in [44, 45]. Indeed, their goal is to track the movements of the same clustered objects over time, e.g., a group of migrating animals. On the contrary, *YouLighter* has no insights about the CDN infrastructure and it cannot track single objects, which may disappear and reappear freely.

Finally, other approaches as [46] measure the similarity among sample distributions obtained at different time intervals. However, directly relying on distributions to perform the comparison considerably complicates the detection of the edge-nodes behind the changes. And from the datamining community evolutionary clustering techniques such as [41, 42] have been used to study how clustering results evolve over time. Instead, *YouLighter* extracts and compares clustering patterns, which are simpler to process in an automatic manner, and allow to immediately pinpoint the edge-nodes (i.e., the clusters) responsible for possible deviations.

A preliminary version of this work has been presented [47]. In this extended version I present more thorough performance analysis and sensitivity study of *YouLighter* algorithms, present a complexity evaluation and discuss possible counter-measures ISP can take to mitigate eventual problems.

2.3 Datasets

I assume the ISP has instrumented each border router of its network with passive probes, which collect statistics from traffic flows carrying YouTube videos. In this work, I rely on passive probes running Tstat as previously explained in Sec. 1.1 ad depicted in Fig.2.1. Among Tstat features I exploit its capability to classify TCP flows that carry YouTube videos. For each request, among all Tstat metrics I use: i) the anonymized client IP address, ii) the server IP address, iii) the hostname of the server as returned by DN-Hunter, iv) the TCP minimum Round Trip Time (RTT),

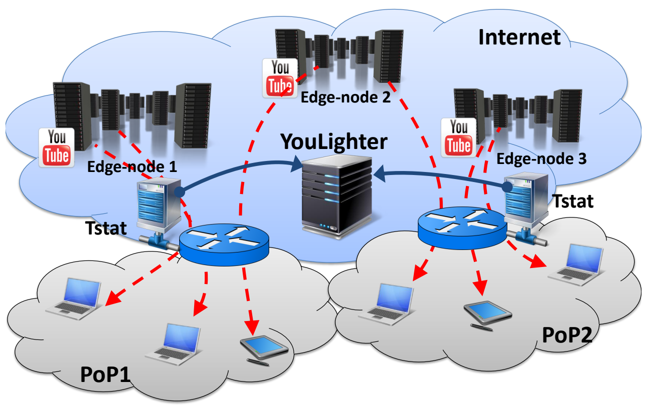


Fig. 2.1 The traffic monitoring setup I employ for this work.

v) the IP Time-To-Live (TTL) of packets received by the client in the PoP, vi) the amount of bytes the clients send and receive, vii) the average download throughput, and viii) the time at which the TCP connection starts.

Since Tstat can compute all these metrics even in presence of encryption, it allows me to take in to account also HTTPS, which nowadays represents more than 50% of overall YouTube traffic [48].

Trace	Period	Volume	Flows	Caches
ISP1-A	01/04/2013 - 28/02/2014	138.7 TB	33,216,794	8,664
ISP1-B	01/04/2013 - 28/02/2014	152.9 TB	31,643,603	8,899
ISP1-C	01/04/2013 - 28/02/2014	134.8 TB	27,377,089	9,028
ISP2	01/03/2014 - 17/07/2014	48.3 TB	9,100,163	3,755

Table 2.1 Traces considered in this study.

I have been collecting traffic logs since April 2013 by monitoring the traffic users generate when accessing the Internet. I instrument four different PoPs. Three of them are located in networks of the same ISP, and in two different cities of the same country. I install the fourth one in a PoP of a different ISP in a second country. Tab. 2.1 describes, for each trace (or dataset), the time period, the total downloaded volume, the number of unique videos and the number of YouTube servers I observe. Notice that in total I monitor the activity of more than 32,000 customers, and the maximum number of YouTube caches that ISP1 customers used at least once is ∼9,000. The dataset overall covers more than 500 TB of equivalent data.

2.3.1 YouTube Cache Naming Structure

I find that the YouTube infrastructure described in [29] is no longer in use. During 2012, YouTube server hostnames were in the form `rx--ABCxxtxx.c.youtube.com`, where `x` are numbers, while `ABC` is a three-letter code reporting the International Air Transport Association (IATA) code of the closest airport. For instance the hostname `r7--fra07t16.c.youtube.com` identified a single cache, in Frankfurt. The hostname still resolves to a single IP address, 74.125.218.182 in the example. Thus, I can uniquely identify a cache by its hostname.[4] All caches co-located in the same edge-node share the same (obfuscated) IATA code. This allows us to get coarse ground truth about the location of servers.

I run some active experiments to cross-check if YouTube specializes caches to serve some particular content, and I verify that every cache can serve any video, at any resolution, in any format, e.g., MPG4 or Flash, to any device, e.g., PC, smartphones or tablets.

2.3.2 Characterization of the Load Balancing Policies

Every time a user starts a video playback, the player starts a progressive download of the video content from the specific cache the system provides in the HTML page.[5] I am interested in seeing which are the policies governing the server allocation, such as (i) is there any "preferred" group of caches? or (ii) are those stable over time?

Fig. 2.2 reports the rank of YouTube caches sorted by the number of flows they serve. I consider February 2014 from the *ISP1* datasets. First, notice that I observe up to 3,800 in *ISP1-C*, with *ISP1-A* reaching more than 2,500. Second, the load each cache handles is very heterogeneous in all datasets; few servers handle lots of requests, but there is a not negligible number of caches that serves a significant portion of flows. In all three datasets, top 400 caches serve more than 1,000 videos,

[4]Starting from January 2013, YouTube obfuscates the IATA code using a simple substitution cipher. The script to de-obfuscate YouTube encrypted cache names is available at http://tstat.polito.it/svn/software/tstat/trunk/scripts/decode_yt_sitename.pl. From October 2013, the `youtube.com` domain has been replaced by the `googlevideo.com` domain. This information can be used to identify YouTube flows even in presence of HTTPS [49].

[5]Load balancing policies are implemented at application layer. Indeed, the web server chooses and encodes the cache hostname directly in the HTML page served to the client.

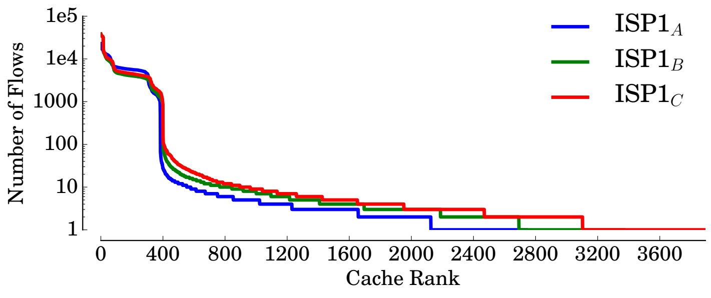


Fig. 2.2 Rank of YouTube caches based on the number of flows. February 2014, *ISP1*.

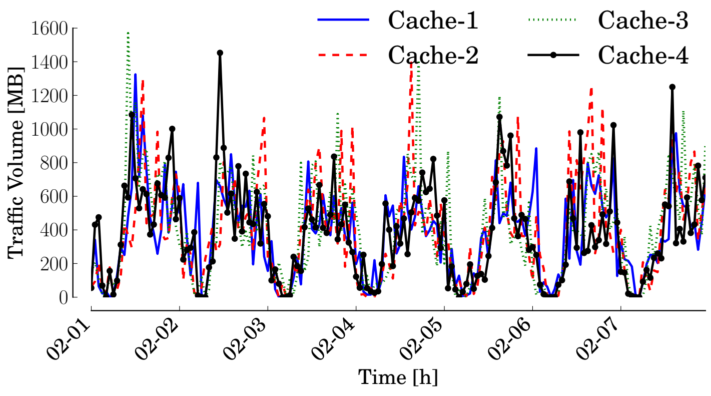


Fig. 2.3 Evolution of the volume of traffic for the four most active caches I observe on February 1st 2014. First week of February 2014, dataset *ISP1-A*.

and, to observe 95% of requests, one must consider about 313, 330, 342 caches in *ISP1-B*, *ISP1-A*, *ISP1-C* respectively.

I also notice that the rank is extremely dynamic over time. For instance, I pick randomly four caches among the most active caches in *ISP1-A* during the 1st week of February 2014. I report in Fig. 2.3 the amount of traffic they generate over time for the following seven days. As shown, the amount of traffic a single cache handles changes widely over time, and none of the monitored caches keeps a constant leading position for a long period of time.

As one may expect this dynamicity to disappear when reducing the focus, I monitor a larger pool of caches as those in the rank in Fig. 2.2, and I recompute the same rank on a daily basis. Then, I represent it using different colors in Fig. 2.4. Each row represents the rank of the same cache for different days in February. In case

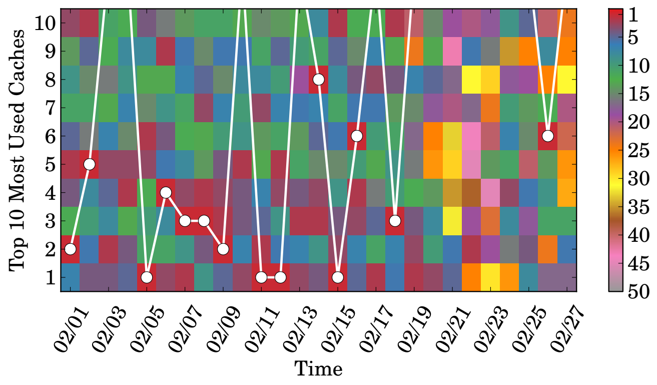
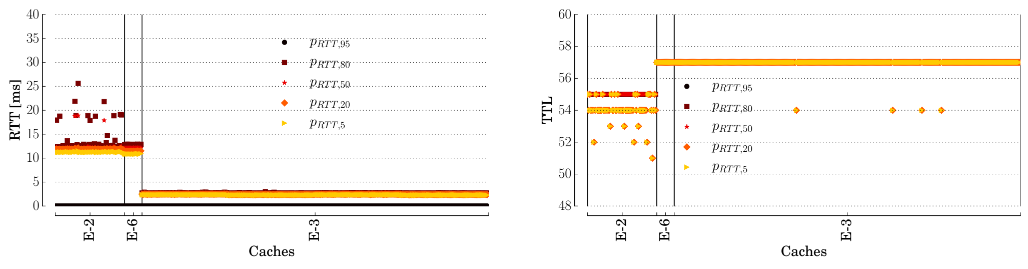


Fig. 2.4 Evolution over time of the rank of the top 10 mostly used caches during February 2014, *ISP1-A*. The white dot corresponds to the top cache of each day.

the rank is stable, one would expect a row (a cache) to always assume the same color (rank). Fig. 2.4 shows exactly the opposite. Indeed the top daily cache (red square, highlighted by the white dot) randomly changes every day (white line). Sometimes, the most used cache in a day is not among the top-10 cache of the month (line jumps outside). The top-10 caches in the monthly rank drops below the 50th place during some days (gray color). Similarly, in the first 19 days of February, the top-10 caches are concentrated in the first 20 rankings; However, starting from February 20th they fall around the 30th position (notice the concentration of yellow and orange boxes).

This shows that the server allocation policies adopted by YouTube spread the load over several hundreds of caches, and the choices are extremely dynamic over time if I observe with the fine grained granularity of a single cache.

Since caches inside the same edge-node are all equivalent, the intuition is to observe the system using the coarse granularity offered by edge-nodes. However, edge-nodes are unknown, they can change over time due to system upgrade, maintenance operations, or redesign. Information that could be available (e.g., the IATA code) may be not reliable, or may be outdated by YouTube. This calls for cognitive systems that can automatically identify presence of edge-nodes and to build a model of the current status of the CDN. I thus design an unsupervised machine learning algorithm to automatically identify edge-nodes from just the observation of traffic flows.



(a) RTT percentiles (b) TTL percentiles

Fig. 2.5 Example of per-cache RTT percentiles and TTL percentiles. Caches sorted by IP address, and grouped by (anonymized) IATA code. December 12-18 2013, dataset *ISP1-A*.

2.4 Methodology

Among cognitive approaches that try to extract models out of data, cluster analysis (or clustering) represents a well known set of unsupervised algorithms that have been successfully used in the literature. Clustering algorithms group objects with similar characteristics [50]. Objects are described by means of features which map each object to a specific position in a hyperspace. The similarity between two objects is based on their *distance*. The closer the two objects are, the more likely they are similar and thus should to be grouped in the same cluster. Typically, the Euclidean distance is used.

The selection of the features plays a key role in the design of clustering algorithms. In this scenario I would like to group together caches based on the network position as seen from a vantage point. Intuitively, the path between two caches in the same edge-node and clients in the same PoP exhibits the same properties, that from a network perspective translates in same RTT and TTL. Conversely, the paths between two caches in two different edge-nodes should present different RTT or TTL. To confirm this intuition I perform the following experiment. I consider three edge-nodes observed in *ISP1-A* during December 12-18 2013. For each cache, I consider all the flow directed to it and I compute the Cumulative Distribution Function (CDF) of the RTT and the TTL extracted from the flows. Then I summarize each CDF with the 5th, 20th, 50th, 80th, and 95th percentiles, and report the result in Fig. 2.5. The x axis reports caches grouped together by (anonymized) IATA code. On the y axis, for each cache, I report the percentile values, shaded with different colors, obtained from the per-cache CDFs. Fig. 2.5(a) and Fig. 2.5(b) reports the results for RTT and TTL, respectively. Since I sort caches based on their IP address and IATA code, caches belonging to the same edge-node appear one close the other. Three edge-nodes are present, E-2, E-6, and E-3. Each hosts a variable number of caches, with E-3 being the largest. As shown in Fig. 2.5(a), the caches in the same edge-node exhibits very similar RTT percentiles, suggesting that I can identify clusters of caches by considering the RTT as a feature. However, E-2 and E-6 exhibit very similar RTT percentile values. This fact clearly complicates the capability of a clustering algorithm to label them as different edge-nodes. Similarly, when focusing on the TTL (Fig. 2.5(b)) I observe that E-6 and E-3 show practically identical values. However, if I put Fig. 2.5(a) on top of Fig. 2.5(b), I can easily distinguish the three edge-nodes as they all exhibit different RTT and TTL combinations, and E-2, E-3 and E-6 emerge clearly as different edge-nodes. This simple example justifies my choice to consider the multidimensional space obtained by combining percentiles for both RTT and TTL features.

I acknowledge that The RTT and TTL samples might be biased by issues due to the presence of congestion in the network path to the caches. However, YouLighter's aim is to capture these kind of events as well, and notify them to the ISP, which then will investigate further to troubleshoot the problem. Anyhow, the correct working point of an ISP network should be far from a regime affected by congestion, so I expect congestion events to be due to actual changes in the CDN infrastructure or in the network paths to the caches, rather than in the ISP network.

2.4.1 Multi-dimensional Clustering

I leverage above intuition to design a clustering algorithm to automatically find homogeneous groups of caches. I use some ingenuity to characterize the path from client to each cache, and then to cluster caches that exhibits similar paths. I can split the process of my methodology into the following steps:

Step 1 - Passive monitoring of YouTube video flows: As described in Sec. 2.3, a passive probe provides the continuous collection of YouTube traffic logs. I log the metadata of each TCP connection, and I store logs in a database for further processing.

Step 2 - Measurement consolidation and filtering: To ease the monitoring procedure, I use a batch processing approach that considers time windows of size ΔT. Thus, every ΔT I generate a "snapshot", and I aggregate and process measurements in it. In the following, I indicate the n-th snapshot as a superscript when needed, e.g., $a^{(n)}$ indicates the metric a at snapshot n. The choice of the time granularity at which a new snapshot is built is driven by the necessity of finding a trade off between the amount of data to consider, and the frequency at which the network administrator is expected to be capable of reacting to anomalies.

I identify each cache x by its IP address. I then group all flows in the same snapshot with the same server IP address to obtain a table where columns correspond to the metric (e.g., RTT, TTL, transmitted packets, etc.), and each row corresponds to a sample, i.e., the tuple of measured values observed within a TCP flow.

Since I am interested in the active caches, I discard those with less than $MinFlow = 50$ samples. I define the whole measurement snapshot n as $X^{(n)}$.

Step 3 - Feature selection and data normalization: Next, I apply a feature selection driven by domain knowledge to select the set $\mathscr{M}$ of *metrics*. In particular, as I am interested in grouping caches according to the path properties, I choose $\mathscr{M} = \{RTT, TTL\}$. Then, for each cache x in the snapshot X, and for each metric $m \in \mathscr{M}$, I generate a Cumulative Distribution Function (CDF). From the distribution, I extract the vector $P_m(x) = \left(p_{m,1}(x), p_{m,2}(x), \ldots, p_{m,k}(x)\right)$ containing k percentiles of m for cache x. Then, I translate the percentile values obtained from the real space to an hypercube space of unitary size. This is a standard approach which allows the data analytics algorithm to work with a fixed parameter configuration. I thus standardize percentiles following a simple normalization:

$$min_m = \min\left(p_{m,i}(x)\ \forall x \in X, \forall i = 1, \ldots, k\right) \tag{2.1}$$

$$max_m = \max\left(p_{m,i}(x)\ \forall x \in X, \forall i = 1, \ldots, k\right) \tag{2.2}$$

$$\bar{p}_{m,i}(x) = \frac{p_{m,i}(x) - min_m}{max_m - min_m} \tag{2.3}$$

Intuitively, Eq.(2.3) normalizes the percentiles of metric m so that $\bar{p}_{m,i} \in [0, 1]$.

At last, $\bar{P}_m(x) = \left(\bar{p}_{m,1}(x), \bar{p}_{m,2}(x), \ldots, \bar{p}_{m,k}(x)\right)$ represents the standardized vector of *features* for the metric m for server x. Recalling that $\mathscr{M} = \{RTT, TTL\}$, I identify each cache $x \in X$ with a $2k$-dimensional which is then normalized in a space of edge 1 and features:

$$\bar{x} = (\bar{P}_{RTT}(x), \bar{P}_{TTL}(x)) \tag{2.4}$$

and I transform the original set of caches X into a set of points $\bar{X} = \{\bar{x}\}$.

Step 4 - Clustering: Among different clustering algorithms I employ the DBSCAN (Density-Based Spatial Clustering of Applications with Noise) algorithm [33] to group together caches based on their multi-dimensional features. I choose DBSCAN because (i) it is able to handle clusters of arbitrary shapes and sizes; (ii) it is relatively resistant to noise and outliers; (iii) it does not require the specification of the number of desired clusters (iv) it is a density based clustering algorithm which perfectly fits the proposed scenario. DBSCAN is a clustering algorithm presented in 1996 [33] which exploits the notion of dense area to clusterize elements. This characteristic is particularly useful in my scenario since caches belonging to the same edge-node should own similar metrics value lying in a small region of the possible space. Even in presence of small difference within the same edge-node, DBSCAN has good performance as it can handle clusters of different shape. Another important point is that DBSCAN requires only two parameters: ε and $MinPts$. ε determines the maximum allowed distance between any given point in a cluster and its closest neighbour belonging to the same cluster, and $MinPts$ the minimum number of points required to form a cluster. Based on that, it classifies all points as being (i) core points, i.e., in the interior of a dense region; (ii) border points, i.e., on the edge of a dense region; or (iii) noise points, i.e., in a sparsely occupied region. Noise points do not form any cluster, while the algorithm puts in the same cluster any two core points that are within ε of each other. Similarly, any border point that is close enough to a core point is put in the same cluster as the core point. The result of this process is a collection $\mathscr{C}$ of clusters $C_j \in \mathscr{C}$, also named as *clustering*:

$$\mathscr{C} = \{C_j\} = \text{DBSCAN}(\bar{X}) \tag{2.5}$$

2.4.2 Highlighting Changes with the *Pattern Dissimilarity*

I am now interested in tracking the evolution of clusters over time, for which, I adapt the methodology presented in [41], as I discuss in Sec. 2.1. As authors presented in [41] compare two clusterings $\mathscr{C}1$ and $\mathscr{C}2$ obtained considering two *different* datasets, i.e., snapshots in my case opens many different scenarios. For instance, i) points that were present in $\mathscr{C}1$ may not be present in $\mathscr{C}2$, and vice versa; ii) points clustered into the same cluster in $\mathscr{C}1$ can now belong to two or more clusters in $\mathscr{C}2$; and iii) the same points that form a cluster in $\mathscr{C}1$ can still form the same cluster, but can be placed in another region in the clustering space in $\mathscr{C}2$. In my case, this corresponds to i) popular caches at snapshot n that are not anymore used at snapshot $n+1$, or ii) some caches at snapshot n that were part of the noise are instead clustered at snapshot $n+1$, or iii) the path to caches suddenly changes at snapshot $n+1$, altering RTT and TTL. Since I do not have any mean to label the clusters in two different clustering results for instance by using a ground truth label. I can not evaluate major changes, i.e., the presence of a totally new cluster by simply evaluating the difference of clustering results by comparing which clusters are shared between the snapshots. Therefore, to evaluate the difference between the clustering, I adapt the historic cost function presented in [41], and I introduce the notion of *Pattern Dissimilarity*.

Clustering Patterns

I first map each cluster into a single *Centroid* that summarizes it. For each cluster, a centroid is computed following the standard approach, i.e., it is the mean position of all the points in all of the coordinate directions. Therefore, given a cluster $C \in \mathscr{C}$, I consider the centroid, or geometric center, $\hat{x}$ whose components $\hat{p}_{m,i}$ in the i percentile of feature m are:

$$\hat{p}_{m,i} = \frac{1}{|C|} \sum_{x \in C} renorm(p_{m,i}(x)) \tag{2.6}$$

All centroids then form a *pattern* $\hat{\mathscr{P}} = \{\hat{x}\}$. The *renorm*() function considers the re-normalization of features that is needed if points in $\mathscr{C}1$ and $\mathscr{C}2$ went through different normalization processes according to Eq.(2.3). In other words, *renorm*() transforms the space defined by the centroids obtained by two different snapshots in a common hypercube space of unitary side. This normalization step is required since normalizing over all snapshots is not possible because in an online analysis system future snapshot are not known a priori. In my case, assuming $\mathscr{C}1 = \mathscr{C}^{(n)}$, $\mathscr{C}2 = \mathscr{C}^{(n+1)}$, from Eq.(2.3) for each $m \in \mathscr{M}$ I have:

$$Min_m = \min\left(min_m^{(n)}, min_m^{(n+1)}\right) \tag{2.7}$$

$$Max_m = \max\left(max_m^{(n)}, max_m^{(n+1)}\right) \tag{2.8}$$

$$renorm_m(a) = \frac{a - Min_m}{Max_m - Min_m} \tag{2.9}$$

To better explain the need of this two-step normalization process I describe a simple one-dimensional example. Consider the case in which the first snapshot has values varying in $[0,15]$, and the second snapshot in $[2,22]$. I first perform a per-snapshot normalization to run DBSCAN, so that I obtain $[0,15] \rightarrow [0,1]$ as by Eq.(2.3). Similarly $[2,22] \rightarrow [0,1]$ for the second snapshot. Next, to compare these two snapshots, the $renorm()$ function will re-normalize the space in a common space with $[0,22] \rightarrow [0,1]$.

Centroid Distance

Given a centroid $\hat{x}$ and a centroid pattern $\hat{\mathscr{P}}$, I define the *Centroid Distance* (*CD*) as the distance between $\hat{x}$ and its closest centroid in $\hat{\mathscr{P}}$. Specifically, I compute the closest centroid $\hat{y}^* \in \hat{\mathscr{P}}$ such that $d(\hat{x},\hat{y}^*) \leq d(\hat{x},\hat{y})\ \forall \hat{y} \in \hat{\mathscr{P}}$. $d(x,y)$ can be any distance metric that is valid in the feature space. In this work, I use the classic Euclidean distance. Thus, the Centroid Distance *CD* of the centroid $\hat{x}$ from centroids in $\hat{\mathscr{P}}$ is

$$CD(\hat{x},\hat{\mathscr{P}}) = \min_{\hat{y}\in\hat{\mathscr{P}}} d(\hat{x},\hat{y}) \tag{2.10}$$

Hence, the Centroid Distance couples centroids according to a nearest neighbor principle.

Pattern Dissimilarity

At last, I define the *Pattern Dissimilarity* - *PD* - as the sum of the Centroid Distance among every centroid in the clusterings. Since the number of clusters in $\hat{\mathscr{P}}1$ and $\hat{\mathscr{P}}2$ may be different, I need to symmetrize the definition:

$$PD(\hat{\mathscr{P}}1,\hat{\mathscr{P}}2) = \sum_{\hat{x}\in\hat{\mathscr{P}}1} CD(\hat{x},\hat{\mathscr{P}}2) + \sum_{\hat{x}\in\hat{\mathscr{P}}2} CD(\hat{x},\hat{\mathscr{P}}1) \tag{2.11}$$

Fig. 2.6 depicts the *Pattern Dissimilarity* computation considering a 2-dimensional space. From left to right, DBSCAN first clusters the points (grey dots for the first

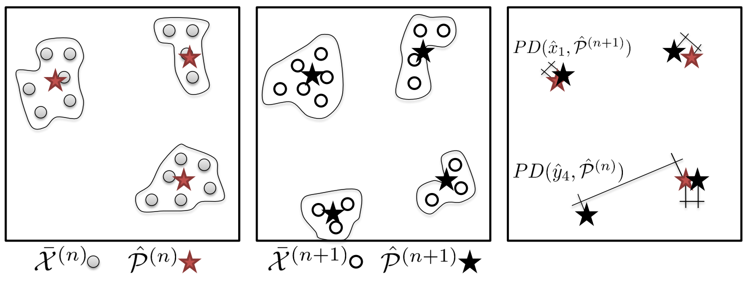

$PD(\hat{x}_1, \hat{\mathcal{P}}^{(n+1)})$
$PD(\hat{y}_4, \hat{\mathcal{P}}^{(n)})$
$\bar{\mathcal{X}}^{(n)}$
$\hat{\mathcal{P}}^{(n)}$
$\bar{\mathcal{X}}^{(n+1)}$
$\hat{\mathcal{P}}^{(n+1)}$

Fig. 2.6 Example of Clusterings, Patterns and Centroid Distance computations.

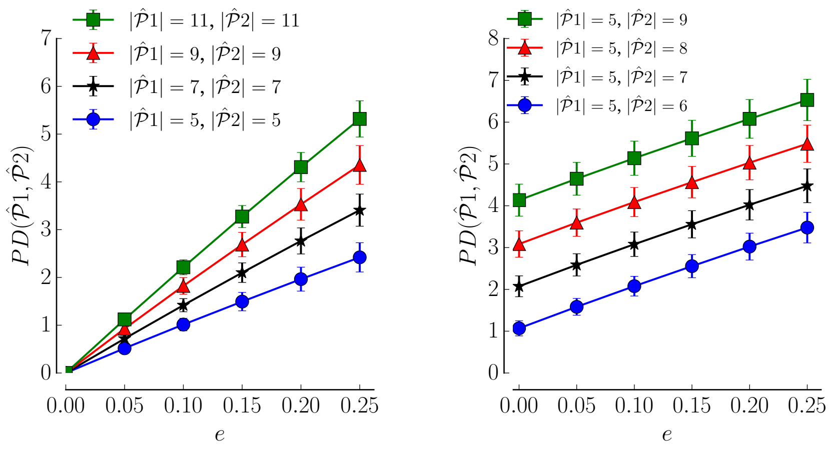
snapshot, white for the second). Then, centroids emerge to form the patterns, and I compute the Centroid Distance for each centroid. Finally, the *Pattern Dissimilarity* is the sum of all Centroid Distances.

In the following, I consider two subsequent snapshots n, and $n+1$, compute the clustering $\mathscr{C}^{(n)}$ and $\mathscr{C}^{(n+1)}$, then extract the patterns $\hat{\mathscr{P}}^{(n)}$ and $\hat{\mathscr{P}}^{(n+1)}$, and finally compute their dissimilarity $PD\left(\hat{\mathscr{P}}^{(n)}, \hat{\mathscr{P}}^{(n+1)}\right)$.

I note that I can base the *Pattern Dissimilarity* on other similarity metrics different from the Euclidean distance, e.g., the well known Cosine Similarity. However, as I show in Sec. 2.4.3 using the Euclidean distance lets the *Pattern Dissimilarity* to inherit linear properties, and therefore to vary proportionally with size of the changes. Observe also that the design of the *Pattern Dissimilarity* offers a nice property that is particularly desirable for troubleshooting purposes. In particular, the *Pattern Dissimilarity*, which is a simple sum of Euclidean distances, lets us immediately pinpoint the centroids responsible for changes in the pattern. As I show in Sec. 2.6, this aspect is crucial, as it allows us to design an automatic procedure that i) captures changes in YouTube CDN infrastructure, and ii) highlights the edge-nodes involved in these changes.

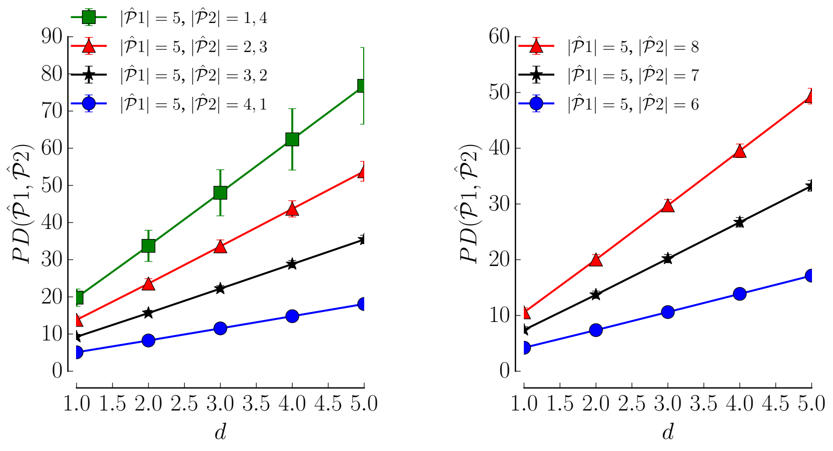
2.4.3 Observations about the Pattern Dissimilarity

I run some numerical evaluation to gauge how the *Pattern Dissimilarity* changes with respect to changes in the input data. I consider two main sources of changes: i) centroids that simply move from their position, and ii) the birth of new centroids reflecting the generation of a new cluster in the data.



(a) $|\hat{\mathscr{P}}1| = |\hat{\mathscr{P}}2|$ (b) $|\hat{\mathscr{P}}1| < |\hat{\mathscr{P}}2|$

Fig. 2.7 *Pattern Dissimilarity* for increasing noise e, for constant and increasing number of centroids.



(a) $|\hat{\mathscr{P}}1| = |\hat{\mathscr{P}}2|$ (b) $|\hat{\mathscr{P}}1| < |\hat{\mathscr{P}}2|$

Fig. 2.8 *Pattern Dissimilarity* for increasing distance among clusters d, for constant and increasing number of centroids.

Changes within the same metric space

Consider first that case where changes to samples leave samples within the same metric space, i.e., $min_m^{(n)} = min_m^{(n+1)}$ and $max_m^{(n)} = max_m^{(n+1)}$.

For the first scenario, I generate a random pattern $\hat{\mathscr{C}}1$ of $N = |\hat{\mathscr{C}}1|$ centroids. I randomly place centroids in the unitary hypercube of edge 1 in $\mathbb{R}^N$ according to a uniform distribution. Then, I generate pattern $\hat{\mathscr{C}}2$ by taking the centroids in $\hat{\mathscr{C}}1$, and repositioning them in a random sphere of radius e centered in the centroid original position. Finally, I compute $PD(\hat{\mathscr{C}}1, \hat{\mathscr{C}}2)$. I repeat the experiment for 100 times, and compute the average and standard deviation of the obtained values. Fig. 2.7(a) reports the average *Pattern Dissimilarity* and its standard deviation for increasing values of e, and for different values of N. As expected, curves pass through the origin, and linearly grow with e. The larger is N, the higher is the average *Pattern Dissimilarity* and its standard deviation.

For the second case, I run the same experiment while also increasing the number of centroids. Thus $|\hat{\mathscr{C}}1| < |\hat{\mathscr{C}}2|$. Fig. 2.7(b) shows the results. Notice the nice property of the *Pattern Dissimilarity* for which the birth of new centroids causes the *Pattern Dissimilarity* to grow by a factor that is proportional to the number of new centroids. This is due to definition in Eq.(2.11) in which no normalization is present. This property is important, as it lets the *Pattern Dissimilarity* nicely highlight the sudden birth (or death) of centroids i.e., edge-nodes in my scenario.

Changes to a larger metric space

Consider now the that case where changes move part of the samples outside the metric space at time n. In particular, let samples move/appear with $max_m^{(n+1)} = dmax_m^{(n)}$, $d > 1$.

Fig. 2.8 shows results. Starting with left plot, I consider the case where 5 clusters are present at both time. At time $n+1$, points of k clusters move in a different portion of the space, being allowed to move is a space of diameter d bigger than the original one. Results show that this change has a much greater impact, see Fig. 2.8(a). In the figure, $|\hat{\mathscr{P}}2| = x, y$ states that x cluster still lie in the original space, and y lie in the upscaled space. Intuitively, since points now move to a larger space, the *Pattern Dissimilarity* of the cluster which lies outside the original space let the *Pattern Dissimilarity* grow higher.

Fig. 2.8(b) consider the birth of new clusters, which lie in the upscaled space of factor d. Also in this case, the *Pattern Dissimilarity* grows much higher than when considering the birth of clusters within the same space – cfr. Fig 2.7(b).

These simple experiment allows us also to select thresholds to highlight important changes, rather than minor changes. For instance, when samples move outside the original space, the *Pattern Dissimilarity* is typically larger than 10. Conversely, when changes move samples within the same space, *Pattern Dissimilarity* is smaller than 10. This will be useful when highlighting significant changes in real data.

2.4.4 Complexity

After evaluating different aspects of the methodology, the last important part to analyze is its complexity. This step is required to understand the feasibility of running *YouLighter* in a real environment. To evaluate *YouLighter* complexity I study the computational cost of the different steps. I assume that the network has been instrumented to extract flow level logs from passive observations of packets using Tstat.

Assume to have N caches and a total number of YouTube flows F. From my dataset, I always obtain N in the order of 1,000 and F amounting to half billion flows in each week of data from ISP1. Therefore, a one week long dataset can be easily stored in memory of MB magnitude. Given the flow level log, the first step consists in measurement consolidation. It consists in grouping flows by cache and filter less used ones. Using an hash-based data structure where the key is the cache IP address, counting flows per cache has a complexity of $O(F)$.

The second step consists in feature extraction and data normalization for each flow. For each cache, empirical percentiles must be computed which entails sorting of flows. Worst case complexity is $O(F\ln(F))$ when $N = 1$ and all flows belong to the same cache. Assuming instead that F flows are equally split among N caches, the complexity becomes $O\left(N\frac{F}{N}\ln(\frac{F}{N})\right) = O\left(F\ln(\frac{F}{N})\right)$.

The third step is running DBSCAN. According to authors in [33], the algorithm complexity is $O(N\ln(N))$. The output consists in M clusters. The final step consists in computing the *Pattern Dissimilarity*. Firstly, it requires to compute the boundary values for the *renorm* function. This operation is linear $O(N)$ since it requires to find the minimum and maximum values of features. Secondly, it computes the *centroid* of each pattern. For each the M patterns, each containing L caches, I have a complexity of order $O(ML)$. Assuming L to be $O(\frac{N}{M})$, the complexity reduces to $O(N)$. The next part has to deal with the centroid distance. This part has a computational complexity of $O(M^2)$. Finally the *Pattern Dissimilarity* is computed with a cost of $O(M)$ since I have to sum all pattern distances. Overall, *YouLighter* complexity is thus dominated by the second step, i.e., $O\left(F\ln(\frac{F}{N})\right)$. For my *YouLighter* implementation, I used *Python* program language which took on average less than two minutes to perform all computation for one week of data.

2.5 DBSCAN performance

In this section I first assess and tune the performance of DBSCAN in order to identify edge-nodes. I next run *YouLighter* over a longitudinal dataset to show its ability to highlight sudden changes in the YouTube CDN.

2.5.1 Clustering Performance Metrics

I first evaluate the impact of the parameter settings on the DBSCAN results. In particular, I aim to understand how good is the matching between the clustering DBSCAN returns and the edge-nodes I observe in the measurements. To perform this analysis, I consider the snapshot X from November 4th to November 10th, 2013, in trace *ISP1-A*. I manually inspect the dataset, and, guided by the IATA codes, I assign each cache a label corresponding to the edge-node in the YouTube CDN. I manually cross-check labels by inspecting server IP addresses and subnets, RTT and TTL distributions to verify the accuracy of the labels. The result is a ground truth label, GT-label, that I assign to each cache. In total I find $|X| = 620$ caches serving more than $MinFlow = 50$ flows, and belonging to 6 edge-nodes, each identified by a different GT-label. Hence, the number of GT-labels is $N_{GT} = 6$.

I then run DBSCAN as described in Sec. 2.4.1, obtaining the clustering $\mathscr{C}$. Let $N_C = |\mathscr{C}|$ be the number of clusters. I next use the GT-labels to assign a label to caches by using a majority-voting scheme: For each cluster $C_j \in \mathscr{C}$, I assign all caches $x \in C_j$ the most frequent GT-label observed in C_j. Caches whose assigned label matches the GT-label are the so called True Positives (TP), whose number is N_{TP}. Conversely, caches whose assigned label is different from their GT-label are False Positives (FP), whose number is N_{FP}. $|X| = N_{TP} + N_{FP}$. I compute the set of distinct labels assigned to clusters in $\mathscr{C}$, whose number is $N_L \leq N_{GT}$. I do not assign any label to the caches which DBSCAN classifies as noise points.

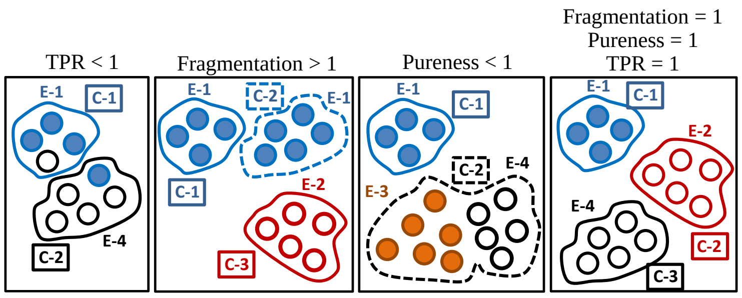


Fig. 2.9 Examples of patterns for which the *True Positive Rate* , the *Fragmentation Index*, and the *Pureness Index* are not equal to 1, and the optimal case in which they are all equal to 1. Color represent the GT-label.

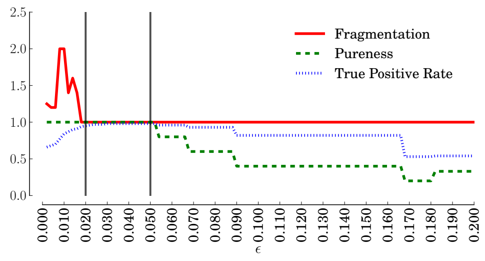
To validate the clustering I obtain with DBSCAN I consider three different indices, as follows:

$$\text{TPR} = \frac{N_{TP}}{|X|},\ \mu = \frac{N_C}{N_L},\ \phi = \frac{N_L}{N_{GT}} \tag{2.12}$$

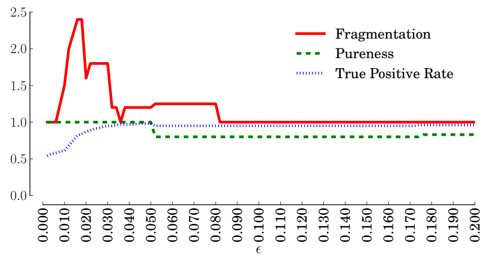
1. The True Positive Rate ($TPR \leq 1$), also known in the literature as “Purity” [50] measures the ratio between TP and the number of samples in the experiment. $TPR = 1$ means that all labels are identical to the GT-label. $TPR < 1$ indicates the presence of i) mislabeled caches (or FP), or ii) noise points (unlabeled points). Leftmost sub-figure in Fig. 2.9 reports a simple example where the clustering algorithm mislabels a cache for both the GT-labels E-1 and E-4, thus leading to $TPR < 1$. Colors represent the GT-label.

2. The *Fragmentation Index* ($\mu \geq 1$) is a custom metric that captures the case when more clusters share the same GT-label. When $\mu = 1$, the number of clusters is identical to the number of GT-labels and DBSCAN assigns each cluster a different GT-label. When $\mu > 1$ instead, I have more clusters which share the same GT-label, i.e., DBSCAN splits an edge-node into two or more clusters. Second sub-figure in Fig. 2.9 reports an example where the clustering algorithm splits edge-node E-1 in two different clusters, C-1 and C-2, thus leading to $\mu > 1$.

3. *Pureness Index* ($\phi \leq 1$) is also a custom metric that measures the ability to identify all edge-nodes. When $\phi = 1$, DBSCAN assigns each GT-label to at least one cluster, i.e., it correctly identifies all edge-nodes. $\phi < 1$ indicates that



(a) DBSCAN with percentiles as features.



(b) DBSCAN with mean and standard deviation as features.

Fig. 2.10 DBSCAN with different feature settings. Performance versus ε. 1st week of November, *ISP1-A*.

some edge-nodes disappear into other clusters (i.e., their GT-label is not the majority label for any cluster). Third sub-figure of Fig. 2.9 reports an example where the clustering algorithm groups together edge-nodes with GT-labels E-3 and E-4 in cluster C-2, thus leading to $\phi < 1$.

Rightmost sub-figure in Fig. 2.9 also depicts the ideal clustering result in which DBSCAN groups correctly the caches for all the edge-nodes, i.e., one cluster for each GT-label (edge-node), leading to the case in which all the clustering performance indices, TPR, μ and ϕ, are equal to 1.

Finally, I use also the number of noise points as an index of bad clustering results, i.e., the inability of DBSCAN to group caches into edge-nodes.

2.5.2 DBSCAN Performance and Parameter Sensitivity

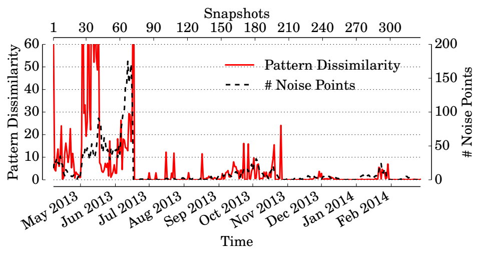
I run experiments to evaluate the impact of DBSCAN parameters, i.e., the choice of the features, $MinPts$ and ε. For now, I set features as the 20th, 35th, 50th, 65th, 80th percentiles for both the RTT and TTL distributions. $MinPts$ is typically not critical since it defines the minimum number of caches in an edge-node DBSCAN needs to form a cluster. I set it to 5. Instead, I must choose ε carefully: If too small, a lot of fragmented clusters will emerge, or a large number of points will not be able to form dense areas, increasing the number of noise points; conversely, large values tend to create few, very large clusters, that aggregates caches from different edge-nodes.

Fig. 2.10(a) reports the clustering indices when varying $\varepsilon \in [0.0 : 0.2]$. As shown, I achieve the best performance with values between 0.018 and 0.052 (in between the vertical solid lines). For such values, all the three indices are equal or very close to 1. Smaller values of ε increase the number of noise points and artificially fragment edge-nodes into multiple clusters. TPR decreases, while μ first increases, then decreases due to caches DBSCAN labels as noise (more than 300 caches fall in the noise for $\varepsilon < 0.005$). For ε larger than 0.052 DBSCAN merges edge-nodes into too few clusters, and both ϕ and the TPR considerably decrease.

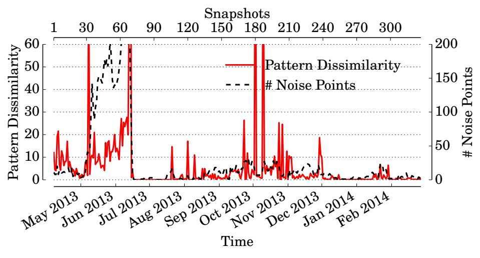
I repeat this analysis for other traces and for different snapshots. I find $\varepsilon \in [0.02 : 0.045]$ to give consistent results. In the following I choose $\varepsilon = 0.04$.

I also run a set of experiments to choose which features to use to capture the RTT and TTL distributions. I replace the vector of percentiles $P_m(x)$ in Eq.(2.3) with simple statistics, e.g., the mean and the standard deviation. The goal of this experiment is to verify whether I can replace the percentiles with some measure which does not require us to build an empirical distribution, a task which requires to collect a fairly large number of flows per cache.

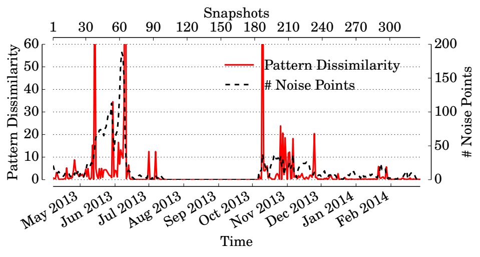
Fig. 2.10(b) depicts results for varying ε. Unfortunately, DBSCAN shows a good clustering for a tiny interval of values of ε, e.g., $\varepsilon = 0.035$. For $\varepsilon > 0.035$, DBSCAN merges edge-nodes together, so that $\mu > 1$ and $\phi < 1$. By investigating further, I observe that the mean and standard deviation vary widely among caches in the same edge-node. This variability is due to the tails of the distributions which include outliers, e.g., very large RTT samples which bias the mean and standard deviation, but have little or no impact on the percentiles. Indeed, the percentiles of caches in the same edge-node are very similar, except those that gauge the tail (see the 95th percentiles in Fig.2.5). This suggests that the choice of the percentiles to populate the vector $P_m(x)$ is more robust with respect to other simpler statistics. I run other experiments with different percentile choices that I do not report for the sake of brevity. I observe no significant differences if I avoid considering percentiles in the tail. Similarly, I observe that using both RTT and TTL gives better results than considering RTT or TTL alone.



(a) *ISP1-B*



(b) *ISP1-A*



(c) *ISP1-C*

Fig. 2.11 *Pattern Dissimilarity* values and number of noise points for different traces from ISP1.

2.6 YouLighter's highlighting capability

In this section I run *YouLighter* over the four traces in Tab. 2.1 to validate its capability of highlighting changes in the YouTube CDN. The rationale is to let the ISP observe macroscopic changes that may affect a large number of users, and which may last for moderate time periods. I consider $\Delta T = 7$ days, and I start a new snapshot at midnight of every day. The choice of the time granularity is driven by the nature of the anomalies that the ISP would like to highlight. Indeed, with too small time scales, e.g., order of hours, more alarms would be possibly raised due to the natural control system of CDNs. At the same time, an anomaly that lasts order of hours would possibly be not relevant for end-users, since the system would return to normality in short time. Snapshots form a sliding window that moves forward every day, and aggregates statistics for the past seven days. $\Delta T = 7$ days guarantees us to obtain a statistically significant amount of feature measurements for each of the caches in the set responsible of carrying the 90% of traffic.

Fig. 2.11 shows the evolution of the *Pattern Dissimilarity* (red solid curve, left y-axes) over time. It also depicts the evolution over time of the number of caches that remain in the noise after clustering (black dashed curve, right y-axes). From left to right, plots refer to *ISP1-B*, *ISP1-A* and *ISP1-C*. X-axes reports daily snapshots, starting from April 1st, 2013.[6]

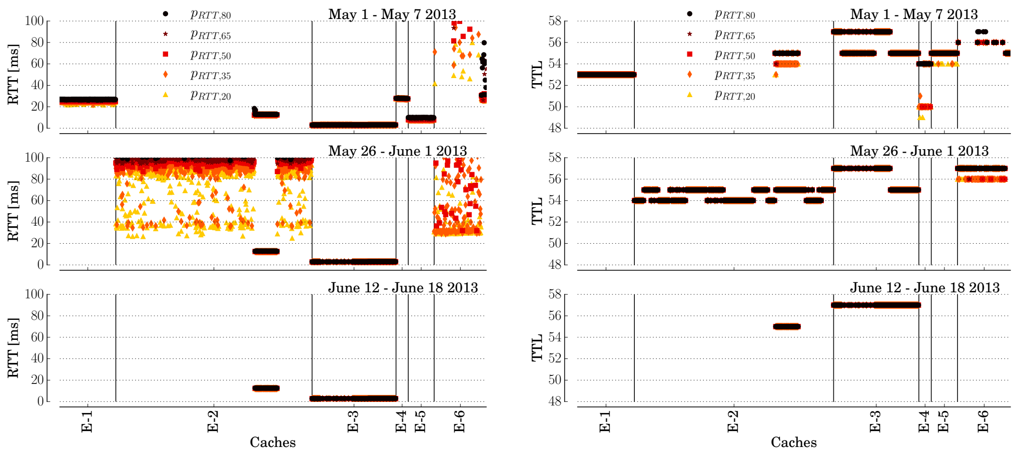
As shown, the *Pattern Dissimilarity* is very good at highlighting events. Indeed, according to Sec. 2.4.3, a $PD > 10$ suggests that the clustering at time (n) is very different to the one at time $(n+1)$. Thanks to the data aggregation I obtain with the clustering, I can easily analyze the highlighted events, and quickly identify the edge-nodes involved in the changes. I investigate these events, and verify that they all correspond to sudden changes in the edge-nodes used by YouTube in serving ISP customers. In the following, I illustrate the most relevant ones, i.e., those with a $PD > 50$.

2.6.1 Large event, involving all ISP customers

I first investigate an event *YouLighter* highlights in three different datasets. It starts on May 2nd (snapshot 27), May 7th (snapshot 32), and May 13th (snapshot 38) for *ISP1-B*, *ISP1-A* and *ISP1-C*, respectively. *Pattern Dissimilarity* peaks above 60. Starting from then, both *PD* and the number of noise points are very large. This indicates an unstable behavior, with many caches that DBSCAN cannot successfully group together, and the clustering pattern that keeps changing day by day, for more than 40 days.

To give the intuition of what happened, left plot of Fig. 2.12 shows the per-cache percentiles of the RTT that I measure in *ISP1-A* before, during, and after the anomalous event. First, I notice that most of the edge-nodes suddenly change: E-1, E-4, E-5 and E-6 actually “disappear” from the clustering pattern, and during the event, many previously unseen caches in edge-node E-2 start serving lots of customers (observe the center plot). Second, and more surprisingly, the path properties to these new caches is by far different from paths to other caches in E-2: the RTT percentiles are much larger (95ms versus 15ms for the 50th percentile) and much more variable. Despite these caches share the same IATA code (E-2), *YouLighter* identifies two clusters since the path to reach caches differs. This is clearly shown by the RTT

[6]PoP referring to *ISP1-C* suffered an outage from mid July 2013 to the end of September.



(a) RTT percentiles (b) TTL percentiles

Fig. 2.12 Per-cache RTT and TTL percentiles during the ISP-wide anomaly in May 2013. Dataset *ISP1-A*.

percentiles, which makes them like to be in two different locations, with the former possibly being severely congested. I call these new cluster Bad-E-2, in opposition to caches showing small RTT, i.e., Good-E-2. While RTT distribution clearly allows us to identify a sudden CDN change, the TTL measurements in the bottom plot of Fig. 2.12 does not reveal any additional information with respect to what RTT already does.

I now analyze the impact of such change on the Quality of Experience the ISP customers perceive. I report in Fig. 2.13 the distributions of the download throughput obtained by video retrieved by caches in E-3, the best edge-node to ISP customers, Good-E-2 and Bad-E-2. The difference is striking: while videos served by E-3 and Good-E-2 have throughput that allows to enjoy YouTube with no major impact on the QoE (>1,000 kb/s in 63% of the cases), the throughput for Bad-E-2 caches is below 500 kb/s (250 kb/s) in 75% (40%) of the cases, clearly not enough to enjoy a video with a satisfiable QoE [51]. Tab. 2.2 corroborates above observation reporting the fractions of video (and audio) formats seen in flows handled by both Good-E-2 and Bad-E-2.[7]. For this analysis I consider only DASH formats, as for these formats the cache delivering the video automatically adapts the quality of the video stream depending on the congestion it measures on the path to the client. As

[7]Observe that in my dataset only a tiny portion (∼1%) of requests are HTTPS, and, thus, encrypted. For the wide majority of the cases, the information about video and audio formats are exposed in plain text in HTTP requests.

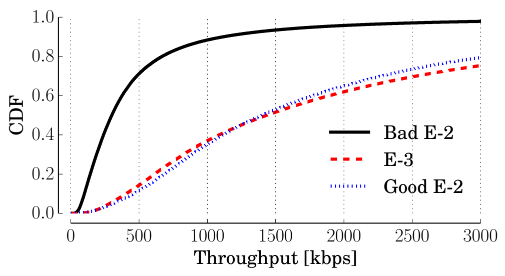


Fig. 2.13 Throughput distribution for flows served by E-3, Good-E-2 and Bad-E-2 during the large anomaly I observe in May 2013. Dataset *ISP1-B*.

Format	**Good-E2**	**Bad-E2**
144p	17.4%	31.7%
240p	18.3%	26.1%
360p	45.4%	35.7%
480p	14.5%	5.3%
720p	3.8%	1.0%
1080p	0.6%	0.2%
AAC128	80.3%	92.0%
AAC256	19.7%	8.0%

Table 2.2 Fractions of video and audio DASH formats served by Good-E-2 and Bad-E-2. Dataset *ISP1-B*.

shown, Good-E-2 serves larger fractions of high-definition videos. Conversely, the share of videos encoded with low-definition (144p and 240p) increases for Bad-E-2. This confirms that Bad-E-2 experienced possible congestion during the monitored period, severely impairing the QoE of the users.

By double checking this event with the ISP network support team, I confirm the incident involved most of their customers, increasing dramatically the complaining at their customer support. This confirms the pervasiveness of this event upon ISP customers.

The accident reported above is an example which testifies that changes in the CDN may raise issues in the video delivery, finally harming users' experience. This highlights how important for the ISP is to monitor and pinpoint changes in the YouTube CDN. The task of measuring how the variability of the CDN structure may impact on the QoE perceived by the users is beyond the focus of this chapter.

2.6.2 Other events for ISP1

I manually cross check other events, and find that some of those affected only part of the ISP customers. This shows that YouTube CDN allocates customers to edge-nodes using a fine grained granularity, i.e., the load-balancing allows to identify small groups of clients by using the client IP address (or network). For instance, on October 2nd (snapshot 180) and October 9th (snapshot 187) *YouLighter* highlights two sudden changes in the *ISP1-A* and *ISP1-C*, as the *Pattern Dissimilarity* peaks

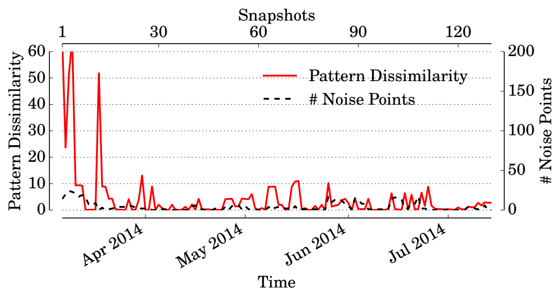


Fig. 2.14 *Pattern Dissimilarity* values and number of noise points for dataset *ISP2*.

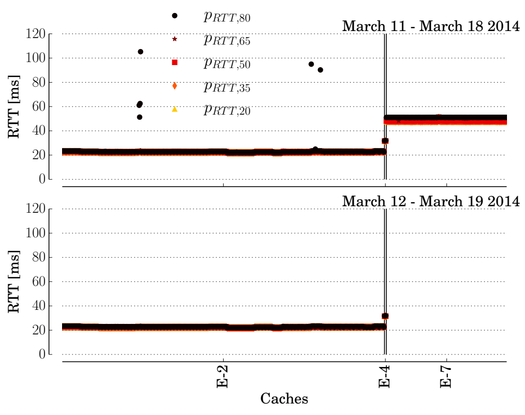


Fig. 2.15 Per-cache RTT percentiles during the second ISP-wide anomaly in March 2014. Dataset *ISP2*.

over 60. Inspecting the astral distances one by one, I observe that the changes are due to 3 edge-nodes (E-4, E-5 and E-6) out of 7 that suddenly "appear" in snapshot 180 and "disappear" in snapshot 187. The remaining four edge-nodes then serve the videos for customers in *ISP1-A* and *ISP1-C*. I analyze the impact of the presence of such caches on the QoE by measuring the aggregate download throughput before, during and after their permanence, but I do not appreciate any significant change. Also in this case I double check the event with the ISP support team and I confirm that the change had no influence on the QoE as the customer support did not receive any meaningful complaining in the considered period.

Finally, observe that for *ISP1-B*, I do not detect any change ($PD = 0.12$) in the same period, as YouTube's CDN keeps serving customers with the same group of edge-nodes, and I do not notice any impact on the QoE for this event too.

2.6.3 Events in ISP2

As a last set of experiments, I run *YouLighter* on the *ISP2* dataset, which I recall I collect in ISP2, a different ISP in a different country. I run *YouLighter* with the same parameters I tune for ISP1, i.e., without going through ε optimization. Indeed I aim to check whether if the edge-node model that DBSCAN creates is general and robust enough to work in a completely different scenario.

I repeat the experiment of Fig. 2.11 for *ISP2* dataset, and I analyze the evolution of the *Pattern Dissimilarity* and number of noise points. I report the results in

Fig. 2.14. To check if the clustering correctly identifies the edge-nodes, I select five different snapshots at random among the ones where *YouLighter* highlights no events. Again, I use the IATA codes as ground truth, and I manually check IP address subnets, RTTs and TTLs to see if some suspicious cache is present in a cluster. The clustering results in perfect match with the (possible) edge-nodes in the ground truth. This despite edge-nodes, path, and ISP in this dataset are completely different.

I then check two suspicious events. The first one occurs from March 7th to March 10th, 2014 (snapshots 1-4, $PD > 60$), and the second one happens on March 18th, 2014 (snapshot 12, $PD > 50$). I observe that the first anomaly is due to a change in the network path to reach a small group of caches in E-2. I observe that this deviation does not influence the QoE perceived by the users. For the second event, by comparing the clustering at snapshot 12 with the following snapshot, i.e., snapshot 13 (March 19th), I observe a notable change in the infrastructure of the YouTube CDN: as depicted in Fig. 2.15 which compares the per-cache RTT percentiles, all caches belonging to edge-node E-7 disappear. Also in this case, the change has no evident impact on users' QoE, as the average download throughput does not vary. However, I notice that the edge-node E-7 represents a much more expensive route for the ISP2, since it is located in an remote ISP for which no peering agreements are in place.

2.7 Countermeasures

In this section, I present some possible techniques that ISPs may consider to this end, leaving a thorough design and analysis for future work.

Firstly, ISPs position allows them to partially control the network routing of traffic. Thus, ISPs may employ traffic engineering techniques to control YouTube traffic using both intra- and inter-routing algorithm. By using BGP, ISPs may reduce congestion by redistributing traffic among different peering links [52]. By announcing via BGP reachability of their network trough other Autonomous System (AS) the ISP can influence YouTube cache allocation policy as well. For instance, ISP1 used this technique to mitigate problems highlighted in Fig. 2.13. In that specific case, ISP1 forced its AS number to be seen as reachable through another ISP (ISP2), located in a different country. This caused YouTube CDN to change the cache allocation policy so that customers from the ISP1 were directed to a edge-node located in the ISP2 country. YouTube traffic was then coming from ISP2 to ISP1, through a high-capacity and uncongested peering link, de facto solving the congestion problem.

A second and more controversial solution would be using DNS directly. By enforcing policies on its DNS servers, the ISP may force a video request to be served by a specific edge-node or even by a specific cache, overruling the decision of the YouTube CDN. As explained in Sec. 2.3.2, as soon as a client requests a video, YouTube selects a cache and returns the cache hostname to the client e.g. `r7--fra07t16.c.youtube.com`. After receiving this hostname the client sends a DNS query to the DNS server (typically managed by the ISP) to retrieve the IP address of the hostname. As a consequence, in case the ISP knows that `r7--fra07t16.c.youtube.com` performance are poor, the ISP DNS can return the IP address of a different cache. This is possible since according to [30, 31], any cache can serve any video. This solution, allows the ISP to select which cache has to manage each request. Therefore the ISP may potentially override YouTube load balancing policies. Some drawbacks have to be faced. First, even if previous studies discovered that any cache can serve any video, the ISP can not be sure that its choice will not introduce a bigger latency due to cache miss, that can cause a further redirection to other caches. Second, the ISP can not control whether a cache will be switched off for maintenance. Observe that an ISP attempt to react to changes in the CDN infrastructure could cause further reactions on YouTube's side too. As such, careful investigations must be done to design and study this kind of policies.

2.8 Conclusions

In this chapter I proposed a novel system, named *YouLighter*, that leverages passive observation of network traffic and unsupervised machine learning techniques to automatically monitor and identify changes in the YouTube CDN. Based on the well known DBSCAN clustering algorithm, *YouLighter* is able to automatically group thousands of caches into few edge-nodes. To then compare the results of clustering obtained considering different snapshots collected in consecutive time intervals, I propose the *Pattern Dissimilarity* which allows to easily pinpoint changes in clusters.

YouLighter is validated using a large dataset of traces reporting the activity of users regularly accessing YouTube. The results are excellent: I show that after a short and simple tuning procedure to find the best setup for DBSCAN, *YouLighter* can detect anomalous events that happened in YouTube CDN. For instance, I could notice a large transformation in a crucial edge-node of YouTube CDN which notably impaired the QoE perceived by the monitored ISP customers for more than 40 days.

I believe that *YouLighter* may represent a promising opportunity for ISPs, network administrators, developers and researchers to monitor the traffic generated by YouTube CDN. ISPs, for instance, may employ *YouLighter* to design automatic traffic engineering policies or to promptly react when changes in YouTube CDN impair the QoE of their customers.

Chapter 3

Exploring Browsing Habits of Internauts: a Measurement Perspective

3.1 Introduction

This chapter presents a paper titled *Exploring Browsing Habits of Internauts: a Measurement Perspective*. This paper has been published in the *Asian Internet Engineering Conference (AINTEC) 2015* conference.[1]

The characterization of users' browsing habits has always attracted the interest of researchers. Since the seminal work of Catledge [53], many studies have been focusing on Web usage, for specific applications [54], or devices [55–57], or at usage of social networks [58]. Understanding how the content is consumed by users is fundamental to improve service design, to offer novel solutions, and, in general, to augment my understanding of the Internet.

In this work, I focus on the exploration of users' habits when browsing the Web. In particular, I take the point of view of the network, from where DNS requests of users are observable, and from which I can extract the *names* of the *services* they are accessing. To simplify the picture, the service is defined by the "second-level domain"

[1]Danilo Giordano, Stefano Traverso, and Marco Mellia "Exploring browsing habits of internauts: A measurement perspective" Proceedings of the Asian Internet Engineering Conference (AINTEC). ACM, Bangkok, Thailand — November 18 - 20, 2015, doi: 10.1145/2837030.2837038

found in DNS traffic, e.g., *google* and *nyt* from *www.google.com* and *www.nyt.com.* I leverage actual anonymized traffic traces collected from operational networks, where 25,000 people access the Web from home. I consider a time window, e.g., one hour, or one day, and I observe how many different websites an Internaut visits, if she keeps discovering new services, or, conversely, if she exhibits repetitive patterns that potentially allows one to build a profile to identify her in a crowd.

I first provide a characterization of the *distinct services* accessed by the entire population. In total, I count more than 400,000 services, with only very few of them being well-known and very popular. Interestingly, the discovery process does not saturate even after two weeks of observation. I next check how the number of *common services* grows over time, i.e., those services that are accessed in multiple periods of time. Intuitively, one would expect that the number of times people access a given service increases for increasing observation period. I instead find out that there is a large number of services that are accessed only once, and by single users. Surprisingly, the growth rates of the two discovery processes (the total number of distinct services, and the total number of common services) is proportional, hinting for a random discovery process.

I then turn my attention on how single users browse the Web, and check if the contacted services can be used to form a fingerprint of each person. I use the Jaccard index to quantify how similar is the browsing performed by i) the same user, and ii) two different users. Intuitively, one would expect that the browsing activity performed by the same user is repetitive over time. Instead, I find limited similarity suggesting a more random exploration of the Web. The affinity among two random users is even more limited, with most of the "common" services being only the most popular ones.

Given this, I run a simple experiment: I test if it could be possible to identify a target user by observing her browsing habits. I extract a user model by monitoring the services she visits during a period of time. Next, I compare the model against 1000 users in a different period of time. The experiment shows that it is possible to identify the target user in less than 50% of cases. This suggests that each individual user has unique interests, but these change over time complicating the task of building a good fingerprint.

3.2 Related Work

In my work I consider how people browse the Web by characterizing the services they access to, as exposed by the DNS names of servers, or hostnames. I rely on DNS information to simplify my analysis, and to extract information from passive measurements also in case encryption is in place. Indeed, while HTTPS is becoming more an more popular thus preventing passive extraction of data from traces, no deployment for encryption of DNS traffic is in place yet. Indeed, DNSSEC [59] provides data integrity and authentication, but no confidentiality, i.e., traffic encryption.[2]

Several works tried to profile users based on their browsing behavior. Authors of [62] tackle the user tracking problem. They exploit HTTP, HTTPS and SSH information to profile and re-identify users over time, using the cosine similarity and clustering techniques based on monthly profiles. Their results show that even by using a month-long profile, the false positive rate is very high 68% for HTTP and 21% for SSH. This confirms the difficulty of tracking user based on their traffic. Similarly, authors of [63] study users' web history collected from browsers to build fingerprints. However, my work differs substantially, as I focus on DNS traffic which, first, can be easily observed from a network layer perspective, and, second, offers a broad view on all the services contacted by the users.

In the field of identification of DNS patterns, authors in [64] study the feasibility of tracking users looking on their DNS traffic. Using a trace collected in a campus network, they evaluate different classification techniques and their effectiveness. Results show that the tracking precision can reach up to 86% by using the Jaccard index as distance metric in a controlled scenario. However, in real case scenarios, accuracy drops to $\approx 50\%$. My work differs as I present a much more detailed characterization of browsing behavior. Moreover, data used in [64] is of 2010, while the Web as evolved significantly. Finally, authors in [65] propose a high level co-clustering algorithm called *Phantom* to clusterize hostnames of servers based on the clients that access them. Differently to my work, they consider classes of services (e.g., e-commerce, news) instead of single services (e.g., ebay, nyt).

[2] There are proposals that address the problem of DNS traffic encryption, the most notable one being DNSCrypt [60]. Their deployment is hampered by the significant infrastructure changes they require[61].

Trace	Period	Users	HTTP(S) flows
VP1	5-18 May 2014	12,262	532M
VP2	7-20 April 2014	13,473	614M

Table 3.1 Details of the datasets considered in this study.

3.3 Dataset

To build the dataset upon which I base my analysis, I rely on a passive probe running Tstat (see Sec. 1.1 for more details), which is installed in the PoP (Points-of-Presence) of the operational network of a national ISP.

For this work, I consider two anomymized 14-day long datasets, *VP1* and *VP2* respectively, collected from two different Vantage Points, located in PoPs in two large cities in the same country. In total, the datasets offer a snapshot of Web browsing activity of more than 25,000 residential customers. Table 3.1 provides details of the datasets. As it can be seen, more than a billion entries are at my disposal.

In the rest of the paper, unless differently specified, I show results for VP1 as VP2 shows similar results.

3.3.1 Data extraction

For my study, I am interested in a specific subset of the TCP information collected by Tstat. In particular, I am interested in the data extracted by DN-Hunter, which is particularly useful for unveiling *services* accessed from simple TCP logs.

Hence, for each TCP flow carrying Web traffic, I extract: (i) the timestamp at which the flow started, (ii) the anonymized IP address of the client, which I employ as user ID hereinafter[3], and (iii) the hostname of the server as recovered by DN-Hunter.

3.3.2 Service definition

The aim of this study is characterizing the Internet users' online activity, and the information they expose. I am thus interested in which "services" users access to, as

[3]The ISP assign static IP addresses to customers' access router, so that the client IP address is a stable and consistent ID.

exposed by the server hostnames. Observe that the hostname offered by DNS is often redundant or not particularly interesting. For instance, consider *www.google.com* or *www.nyt.com*. Clearly, the so called "second-level domain" is the most significant part, e.g., *google* and *nyt*, which is the one that identifies the service being accessed. Similarly, *www.google.de* or *www1.nyt.com* offer little or no additional information. Hence, I consider only the second level domain to identify the services in the following.

Some considerations hold. First, when a user accesses a website, the browser generates a lot of different requests, one for each object composing the webpage. Most of these objects are being served by servers which are not related with the service the user is interested into, but they are used to support the page delivery. For instance, Content Delivery Networks (CDNs), or advertisement platforms, or video streaming caches are regularly contacted when browsing a webpage. These are *support services* and do not identify *actual services*, i.e., the name of the service the user is interested into. Unfortunately there is no easy way to identify them, so that I cannot actually filter them. Second, the frequency with which these support services are contacted is very high [66]. As I will show in Sec. 3.5, the most popular services are indeed support services, which however do not characterize the actual browsing habits of a user. In section 3.5, I provide a characterization of their impact.

If not otherwise stated, the results presented in the remainder of the paper are obtained from the analysis of dataset *VP1*. However, all presented experiments have been conducted on *VP2* too, and I could not observe any significant difference.

3.4 Methodology and Metrics

In the following, I provide a formal description of the methodology upon which I base my analysis. I follow a simple approach based on set theory. Given a flow i, generated by user u_i at time t_i and accessing service s_i, and considering a time interval of duration ΔT, I define the set of services S a user u accessed in the time interval starting from t_0 as

$$S(u,t_0,\Delta T) = \{s_i \mid t_0 \leq t_i < t_0 + \Delta T,\ u_i = u\} \tag{3.1}$$

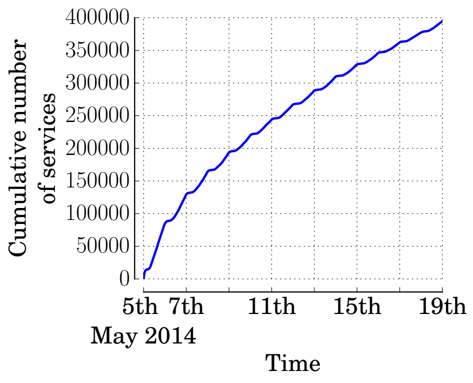


Fig. 3.1 Cumulative number of distinct services over time, *VP1*.

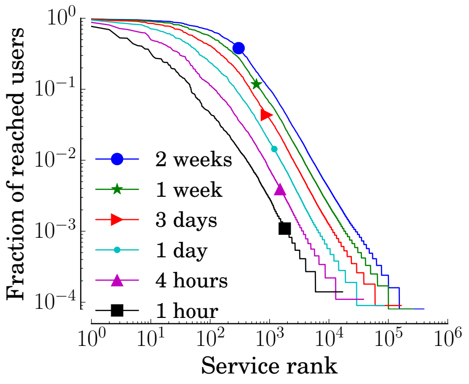


Fig. 3.2 Service rank based on popularity among users, *VP1*.

Similarly, I can define the set of services accessed by all users during a period of time as

$$S(*,t_0,\Delta T) = \{s_i \mid t_0 \leq t_i < t_0 + \Delta T\} \tag{3.2}$$

Let $|S|$ be the number of elements in the set. I can define the *similarity* between the two sets by computing the Jaccard index [67] as

$$Sim(S1,S2) = \frac{|S1 \cap S2|}{|S1 \cup S2|} \tag{3.3}$$

It returns the ratio between the number of common elements over the total number of distinct elements in $S1$ and $S2$. *Sim* equals 0 if there is no common element. If $S1$ and $S2$ contain the same elements, *Sim* equals 1.

Note that I do not consider the frequency with which a service is accessed, but only its presence in the set. Indeed the frequency is highly skewed because of the presence of support services. As I show in the following section, these are contacted by users' browsers much more frequently than actual services.

3.5 Aggregate Characterization

In this section I present measurement results that are useful to understand how an aggregate of ≈12,000 Internet users from *VP1* accesses the Web. This helps in characterizing the dataset size and growth in time.

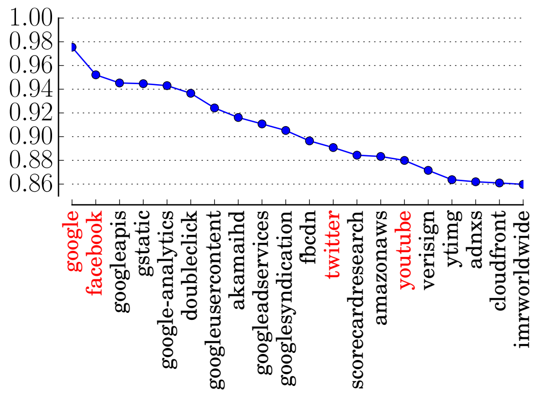


Fig. 3.3 The 20 most contacted services in *VP1*. Actual and support services in red and black, respectively.

3.5.1 Service characterization

Starting from $t_0 =$ May 5^{th}, Figure 3.1 plots the cumulative amount of distinct services I observe over the 14 days of time, i.e., $|S(*,t_0,\Delta T)|$ for increasing ΔT. In total I count about 400,000 distinct services. Interestingly, I observe that, despite I focus on second level domain name only, the number of newly accessed services keeps increasing over time, and does not saturate over the considered period. This reflects the humongous "catalog" of services that people can access on the Internet. The curve exhibits a daily pattern that reflects the typical day/night activity users follow when browsing the Web.

I next analyze the popularity of the services and how it changes when increasing the observation window. Figure 3.2 reports, for several observation intervals ΔT, the service rank according to their normalized popularity, which I compute as the fraction of users accessing a given service among the population of active users at the considered time interval. Each curve corresponds to a different time scale. In particular, the two bottom curves report the ranks computed on one-hour and four-hour long activity periods at peak time for the first day of my dataset, i.e., for $t_0 =$ May 5^{th} at 8pm, $\Delta T = 1h$ and $\Delta T = 4h$ respectively. For the remaining periods, $t_0 =$ May 5_{th} at 0am, and I consider $\Delta T = 1,3,7,14$ days. Notice the log-log scale. Independently on the time scale, all curves present a Zipf-like distribution, with a few services being very popular, and the vast majority of them being contacted by a small number of users. Zipf's distribution is known for governing many aspects of the Internet [68]. It entails that the popularity of services quickly decreases, so that the majority of them are actually contacted by a handful of people, and only few services can reach large popularity. Indeed, considering the $\Delta T = 24$h, only the top 0.26% services are contacted by more than the 10% of active users, and only ≈36% of services are contacted by more than one user. The shift of the curves toward the right part of the plot reflects the growth in the number of *distinct services* seen in Figure 3.1. Conversely, the shift toward the top of the plot reflects the growth in the number of active users. Finally, the shift of the knee toward the upper right part of the plot reflects the increase of *common services*.

Most of the common services are actually support services. To show their impact, I focus on the service popularity rank computed considered $\Delta T = 1$ week, and detail the top 20 most popular services. I report them in Figure 3.3. Names in red correspond to *actual services*, while names in black are *support services* which the browser contacted, but which the user did not explicitly access. Interestingly, among the 20 top popular services, only four, i.e., *google*, *facebook*, *twitter* and *yahoo*, are those expected to be actually requested by the users. Others are CDNs (*akamai*, *googleusercontent*, *fbcdn* - the Facebook CDN, *ytimg* - the YouTube CDN serving images, *cloudfront* - the Amazon CDN), or advertisement and user-tracking platforms (*google-analytics*, *doubleclick*, *googleadsservice*, *googlesyndication*, *scorecardresearch*, *adnxs*, *imrworldwide*), or cloud service (*googleapis*, *gstatic*, *amazonaws*, *verisign*). Investigating further, the vast majority of the services a user encounters during her online activity are *support services*, and this is nicely reflected in DNS traffic. Those are among the most popular services, as clearly shown by Figure 3.3. However, those are only a fraction of the services users access in the Internet. For instance, by randomly picking 100 services, and manually checking them, I count approximately 15% were support services, with 85% being actual services.

Next, I conduct a simple experiment to calibrate the parameter ΔT described in section 3.4. I divide the dataset in smaller portions of duration ΔT. Hence, I obtain time bins aggregating the services contacted by all users, and that I represent as $S(*,t_i,\Delta T)$ for which $i \in 1,...,\frac{L}{\Delta T}$, where L is the total duration of the dataset. Then, I consider each bin pair
$(S(*,t_i,\Delta T),S(*,t_j,\Delta T)),\forall i,j \neq i$. I define the set of services which are present in

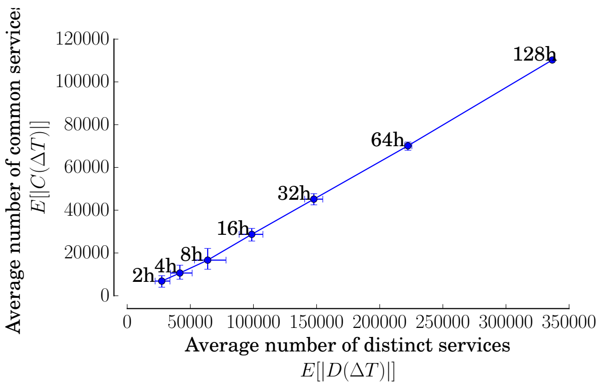
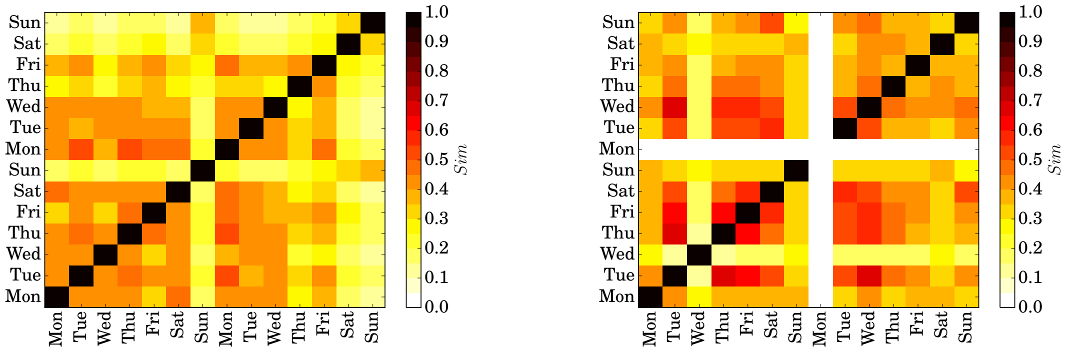


Fig. 3.4 Average number of distinct services, $E[|D(\Delta T)|]$ vs the average number of common services, $E[|C(\Delta T)|]$, for $\Delta T \in [2,4,8,16,32,64,128]$ hours, *VP1*.



(a) **Very active user**. (b) **Moderately active user.**

Fig. 3.5 Heatmaps reporting the matrices of similarity calculated across 24-hour long bins for two example users.

both bins, i.e., the *common* services, as

$$C(t_i,t_j,\Delta T) = S(*,t_i,\Delta T) \cap S(*,t_j,\Delta T) \ \forall\ i,j \neq i, \tag{3.4}$$

and I compute the average number of common services over all possible pairs for a given ΔT as

$$E[|C(\Delta T)|] = \frac{\sum_{i,j\neq i} |C(t_i,t_j,\Delta T)|}{\left(\frac{L}{\Delta T}\right)\left(\frac{L}{\Delta T}-1\right)} \tag{3.5}$$

For each bin pair I also compute the union, thus obtaining the set of *distinct* services appearing in the two bins as

$$D(t_i, t_j, \Delta T) = S(*, t_i, \Delta T) \cup S(*, t_j, \Delta T) \; \forall \; i, j \neq i \tag{3.6}$$

and I compute the average number of distinct services over the number of pairs for ΔT as

$$E[|D(\Delta T)|] = \frac{\sum_{i,j \neq i} |D(t_i, t_j, \Delta T)|}{\left(\frac{L}{\Delta T}\right)\left(\frac{L}{\Delta T} - 1\right)} \tag{3.7}$$

I now compare the growth of the number of common services to the growth of the number of distinct services for increasing ΔT. One would expect that the number of common services would grow faster, since, for increasing time, the chance that a service appears in two different snapshots of time is higher. Figure 3.4 shows the results. It reports $|C(\Delta T)|$ versus $|D(\Delta T)|$ for different values of ΔT. Blue bars report the 20^{th}- and 80^{th}-percentile of the distribution among all pairs. Surprisingly, observe how the growth of number of common and distinct services is linearly proportional, i.e., the number of common services grows with a rate that is proportional to the growth of the number of distinct services. The ratio among them is 1.44. This entails that the common services are approximately 35% of distinct services, independently with respect to the size of the observation window.

In summary, the catalog of services in the Internet is very large. People keep accessing previously unseen services, so that the total number of distinct services keeps growing over time. Surprisingly, the chance that one visits a service that has been already visited or a new service does not depend on the observation interval. Given this, for the experiments presented in the remainder of the paper I choose ΔT =24 hours.

3.5.2 Population characterization

I now focus on observing the users' activity. In particular, I am interested in characterizing users based on the amount of services they contact, to check for instance for the presence of heavy-hitter users, and occasional users. For each user, I count the number of visited services, considering one day. Results, not shown here for lack of space, show that in a single day 50% of users contacts less than 100 services, with only 15% of the most (least) active users that contact more (less) than 250 (30) services.

I leverage this distribution to arbitrarily define three classes of users upon which I will base the experiments presented in the following. First, I define as "inactive" the users who contact less than 50 services per day. For instance, for the first day, these represent 36% of users. I next pick 1000 "moderately active" users, set $\mathscr{MA}$, as those users who contact from 650 to 750 services over the whole trace period.[4] Finally, I consider the top 1000 most active users, i.e., those with at least 1200 contacted services, which I call "very active", set $\mathscr{VA}$.

3.6 Characterization of Users' Habits

In this section I employ the Jaccard index to first observe how repetitive is users' browsing by comparing services contacted during different time periods. Then, in the second part of the section, I compare browsing habits among different users.

3.6.1 Are user's habits similar over time?

I aim at gauging whether users tend to be repetitive in their browsing activity. To this end, I consider all users in the very active and moderately active classes. For each user u, I build the subsets containing the services corresponding to their activity in different time period t_i of duration ΔT=24 hours. For each user, I obtain 14 independent sets, $S(u,t_i,\Delta T)$, one for each day in the trace. Next I compute the similarity index across all set pairs

$$Sim(u,t_i,t_j) = Sim(S(u,t_i,\Delta T),S(u,t_j,\Delta T))\ \forall i, j \neq i. \tag{3.8}$$

To provide an intuitive representation of the outcome of the experiment, Figure 3.5 reports two examples of users I randomly picked in the very active (top) and moderately active (bottom) classes. For each pair of days (t_i,t_j), it reports $Sim(u,t_i,t_j)$ using a color map; The darker the color, the higher is the similarity. Days start from Monday the 5^{th} of May, from the bottom left of the plot.

[4] I chose these thresholds to be just above the median, so that I am relatively confident to focus on a set of consistently active users.

In general, I observe that the similarity is fairly low, with most of values that are smaller than 0.5. For the active user, 3.5(a) the similarity index decreases during the weekends (notice the yellow columns on Sundays). This corresponds to the user not heavily browsing during the weekend, but leaving some device connected to the network, and periodically polling web services, e.g., looking for software updates, or syncing with cloud services. As such, there is a common substrate of support services that are contacted, and that generate a minimum amount of similarity.

Consider the case of the moderately active user, 3.5(b). Some days present a very high similarity, while others seem lightly similar. The white bar on Monday is due to the fact that the user contacted less than 50 services during that day, thus falling in the "inactive" group for which I do not compute the similarity at all. In general, it is hard to identify a regular pattern. For instance, the activity the user performs on Wednesday, first week, is very dissimilar to any other day in the dataset. Or conversely, Tuesday of the first week, and Wednesday of the second week exhibit a very high similarity. Yet, the same Tuesday results rather different when compared to the following Wednesday.

3.6.2 Impact of the number of contacted services

Next, I explain how the properties of the considered time periods impact the similarity index $Sim(u,t_i,t_j)$. In particular, I investigate the effect of the number of services found in the two time intervals. To this end, for each user u in the moderately active class $\mathscr{MA}$, and for all pairs (t_i,t_j), I compute the similarity $Sim(u,t_i,t_j)$. Let $m = min\left(|S(u,t_i,\Delta T)|,|S(u,t_j,\Delta T)|\right)$ be the minimum number of services in the subsets of services contacted during the considered time intervals. I plot $Sim(u,t_i,t_j)$ versus m for all possible users u and all possible pairs (t_i,t_j).

The result is depicted as a scatterplot in Figure 3.6. Curves reports the average (solid red curve) and median (dashed red curve) similarity for samples that fall in bins of size 10, i.e., where of $10k < m < 10(k+1), k = 1,2,\ldots,30$. Observe that when the minimum number of services is small ($m < 50$) the similarity is on average very small, and varies widely, reaching 1 for very small values of m. This is due to time period pairs in which the number of contacted services is very low, and their intersection contains very popular services such as, e.g., *google* or *google-analytics*, the majority of which are support services. This further justifies the choice of not

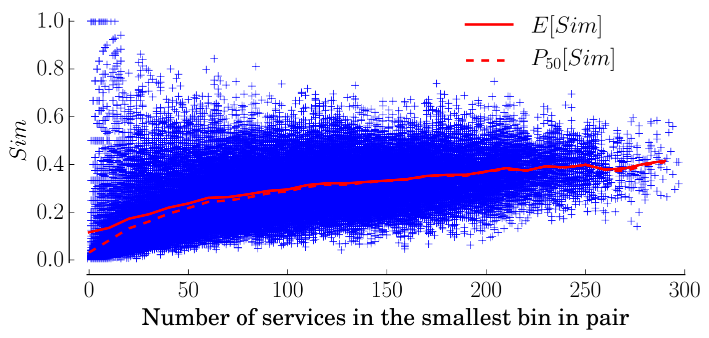


Fig. 3.6 Per-bin-pair similarity index vs the minimum number of services, *VP1*.

computing the *Sim* index when the minimum number of services contacted in one of the two bins is below 50. Conversely, as the number of services grows, the similarity score becomes higher, and at the same time the maximum observed score decreases. Looking at the mean and median values, both stabilizes to a value of ≈ 0.4. Conversely, both are below 0.25 when the minimum number of services is smaller than 50. This is explained by the fact that, in general, sets with a small number of services are compared against sets with large number of services. By definition of the Jaccard index, the denominator (the number of element in the union) has a size which is much larger than the numerator (the number of elements in the intersection).

I complement above observations with another experiment. Again, for each moderately active user $u \in \mathscr{MA}$ I collect the number of contacted services in each time period t_i, and I compute the number of elements it contains, i.e., $n_i = |S(u,t_i,\Delta T)|$. I then consider all pairs (t_i,t_j) $i \neq j$, and consider a discretised grid counting the number of services in buckets of size 10. In other words, I count the number of pairs falling in each bucket. Intuitively, this shows the number of pairs having a given number of services in each. The result is a symmetric matrix, depicted in Figure 3.7. Darker colors are assigned to buckets containing larger values elements. The plot shows that a considerable fraction of pairs falls in the area where the number of services is smaller than 50, with a very large number of pairs that falls in the (< 10, < 10) bucket (observe the dark block in the origin). These are time periods during which the user is actually inactive, and during which mostly support services are contacted. As explained above, when the number of services is so small,

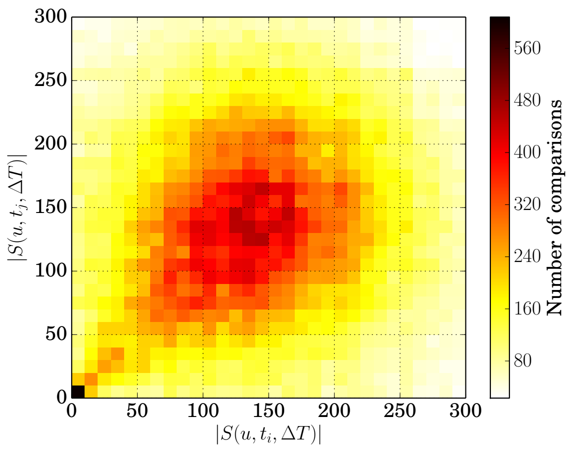


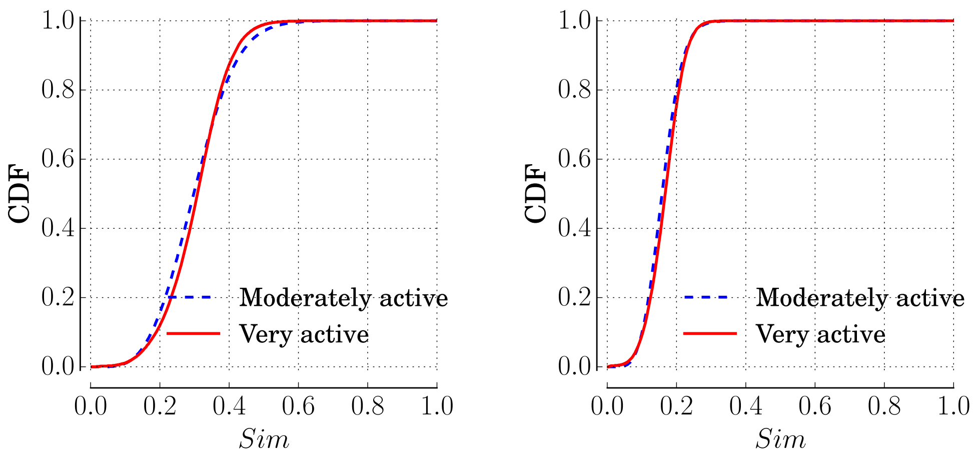
Fig. 3.7 Number of pairs (t_i,t_j) with a given number of service in each.

the similarity index computation leads to very variable results, and thus I prefer to not compute it.

Observing Figure 3.7, I notice that the largest portion of pairs falls in the area between 70 and 220, i.e., where the number of pairs is large, and thus allows us to obtain a significant similarity measure.

3.6.3 Self similarity

In this section I analyze how habits of the same users look similar to each other. I take each moderately and very active user separately, and for each of them I compute the $Sim(u,t_i,t_j)$ index across different time periods (t_i,t_j) for which they results active, i.e., $n_i \geq 50$, $n_j \geq 50$. Then, for each class of user, I build the distribution of the similarity values. Results are depicted in 3.8(a). First, observe that similarity is higher than 0.1, and it saturates at $Sim = 0.7$. This means that, independently of the volume of their activity, the sets of services contained in time bins are significantly different. In other words, users tend to contact same services over time, but the number of new services is however fairly large, meaning that the degree of repetitiveness is low. No significant difference is observed comparing very active and moderately active users.



(a) **Same users.** (b) **Different users.**

Fig. 3.8 Distribution of the similarity index computed across time bins belonging to the users in very active and moderately active classes.

3.6.4 Similarity across users

In this section I analyze how habits of different users look similar to each other. Again, I consider the services contacted by users in different time periods. I calculate the *Sim* index across different users, and I build the distributions for the values obtained separately for the very and moderately active users. I report the results in 3.8(b). As shown, the similarity among different users is very small, and smaller than the one among the same user, cfr. 3.8(a). Indeed, CDFs now saturate at $Sim = 0.3$, meaning that the activity of different users is in 70% of cases very different. As above, no significant difference is observed when considering very active or moderately active users.

This may induce one to conclude that a user is easily identifiable by the set of services she contacts. I next run a simple test to check weather this intuition is correct or not.

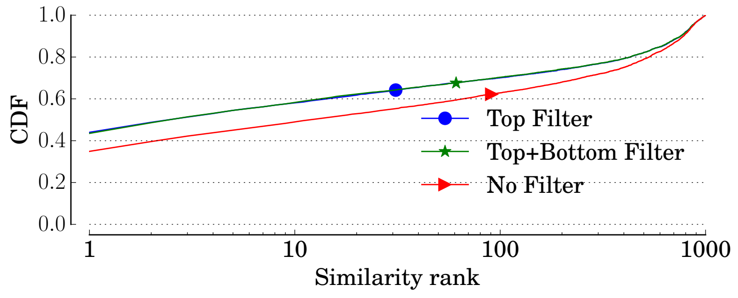


Fig. 3.9 Distributions of the number of times a user is found to be the X^{th} most similar to herself across different observation windows for three different filters.

3.7 Building a simple user classifier

I run now a simple test to check the capability of correctly identifying a user based on the set of services she contacts. To this end, I focus on the moderately active user class. I use the datasets I obtain from the fist day of *VP1* to build a user signature, and then check if it is possible to recognize her in the subsequent days. In more details, for each user $u \in \mathscr{MA}$, I extract the set of services it contacts on the first day t_0, $S(u,t_0,\Delta T)$. Next, I compute the similarity index against, $S(u_i,t_j,\Delta T), \forall\, u_i \in \mathscr{MA}$ and $j \in \{1,...,13\}$, i.e.,

$$Sim(u,u_i) = Sim(S(u,t_0,\Delta T),S(u_i,t_j,\Delta T)). \tag{3.9}$$

I then rank users for decreasing similarity. Intuitively, I want to check how many times the most similar user turns out to be u in a group of a 1000 users $\{u_i\}$.

In my experiments I observe that the choice of the services upon which I compute the similarity is crucial. For instance, I have seen in section 3.5 that the top popular services tend to be support services that users connect to, but do not explicitly browse. Since those are extremely popular, the chance they are in the common set is high, but they do not characterize the user behavior. Similar, those services that do not belong to the common service group are by definition accessed once by one user. Those again would not contribute to identify the user behavior in a different time period. Hence, I create three simple policies to filter the services to build the sets used to characterize the user:

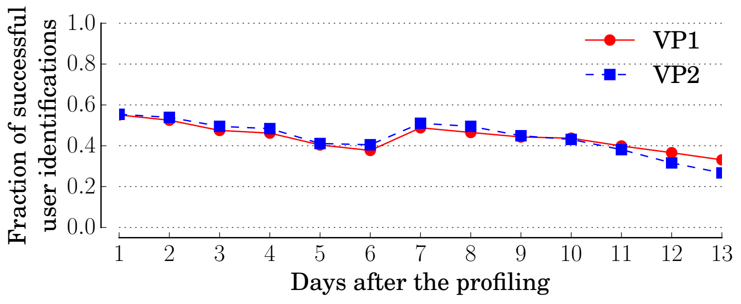


Fig. 3.10 Fraction of times users are successfully identified by their profile at day t_0 over the subsequent days.

•**No Filter**: no filter is applied and all services are considered when building $S(u,t_i,\Delta T)$.

•**Top Filter**: I filter out the most contacted services, i.e., those contacted by more than 50% of users considering the whole dataset.

•**Top+Bottom Filter**: I filter the most contacted services considering the whole dataset, plus those accessed only once in each $S(u,t_j,\Delta T)$.

Figure 3.9 depicts the probability that the user u is found to be among the X^{th} most similar users to herself in the following days, being X the position in the similarity rank. For instance, the leftmost point reports the probability that the user is found to be the first in the rank, i.e., $argmax_{u_i}(Sim(u,u_i)) = u$.

As shown, when no filtering is applied, in 36% of the cases I can successfully identify a user. Conversely, in 50% of the cases a user has at least other 9 users who exhibit a higher similarity index than herself. The Top and Top+Bottom filters achieve better results, but the chances for a user to be uniquely identified increase only to 44%. All these observations demonstrate that users' browsing habits are only partially repetitive enough to allow one to easily build a model of the user so that later would it be possible to identify it in a population of Internauts. My results confirms previous finding [64].

3.7.1 Time stability of the model

I repeat the above experiment considering how similar the results are when comparing sets obtained from far apart days. For a user u, I build a model considering the services she visits during the first day, i.e., $S(u, t_0, \Delta T), t_o =$ May 5^{th}. I then compute the similarity for all days in the following two weeks among all $u_i \in \mathscr{MA}$. For this experiment I employ the Top-Bottom filter. Figure 3.10 shows the probability to correctly identify a user u in the next days, i.e., when $Sim(u, u, t_j) > Sim(u, u_i, t_j)$. I report results for *VP1* and *VP2* to show how similar are results. Interestingly, I observe that the probability to correctly identify the same user decreases over time. This hints that users tend to visit different services depending on the day of the week, and their online activity slightly differentiate over time. However, observe how after one week the similarity index increases again. This suggests that users tend to exhibit a weekly periodicity when browsing the Web.

3.8 Conclusions

In this chapter I leveraged an actual dataset I obtained from an ISP to analyze how Internauts browse the Web by using simple DNS traffic.

Similarly to other studies in the literature, I observed that few hundreds of services are contacted by a significant fraction of users, but only a few intentionally accessed. Indeed, a large fraction of services like CDNs or online trackers are automatically contacted by users' client, and this is nicely reflected at DNS level. I have also shown that users keep visiting and discovering novel services, so that the total number of distinct services keeps growing. Interestingly, I observed that the number of new distinct services grows with a rate that is proportional to the growth of the number of services contacted by multiple users.

When trying to evaluate if the behavior of users follow a periodic and unique profile, I have seen that in practice the similarity of the browsing activity during two different periods of time is rather limited. This surprisingly entails that users are not repetitive in their browsing. However, the practice of building user profiles based on fingerprints made by services contacted is very unreliable. Indeed, in my experiment I could uniquely identify users based on their activity in only 44% of the cases. Moreover, I observed that the profiling effectiveness reduces over time.

Chapter 4

A First Characterization of Anycast Traffic from Passive Traces

4.1 Introduction

This chapter presents a paper titled *A First Characterization of Anycast Traffic from Passive Traces*. This paper has been published in the *The Traffic Monitoring and Analysis (TMA) 2016* workshop.[1]

IP anycast allows a group of geographically distributed servers to share a common IP address. When a client contacts an IP anycast server, the packets are thus routed at the network layer to the closest address according to the BGP routing distance.

Deploying an IP anycast service is relatively easy and introduces several advantages such as load balancing between the servers, DDoS mitigation, and increases the reliability. This is the primary reason of its adoption in multiple *stateless* services running on the top of UDP, e.g., DNS root and top level domain servers, 6-to-4 relay routers, multicast rendezvous points, and sinkholes.

When considering *stateful* services, the usage of IP anycast has been discouraged primarily due to its lack of control and awareness for the server and network load. Indeed, IP anycast relies on BGP routing, meaning that any routing change could

[1]Danilo Giordano, Danilo Cicalese, Alessandro Finamore, Marco Mellia, Maurizio Munafò, Diana Zeaiter Joumblatt, and Dario Rossi. "A First Characterization of Anycast Traffic from Passive Traces" Proceedings of the The Traffic Monitoring and Analysis (TMA). Louvain La Neuve, Belgium — April 7 - 8, 2016, http://tma.ifip.org/2016/papers/tma2016-final30.pdf

re-route the traffic to another server. This could break any stateful service, e.g., causing the abortion of TCP connections, and could cause the dropping of any application layer state. Moreover, the relatively slow convergence of routes and the purely destination based routing in IP make difficult design reactive systems where traffic can be arbitrarily split among multiple surrogate nodes. This initially led the Content Delivery Network (CDN) companies to use other load-balancing techniques, i.e., leveraging DNS or HTTP to direct the customer requests to the best surrogate server [69]. Over the years however, several studies proposed techniques to overcome these issues, showing that it is possible to leverage anycast for connection oriented services [70–72]. This let companies to deploy Anycast-enabled CDNs (A-CDN), in which multiple surrogate servers use the same IP address whose reachability is managed by IP anycast. CacheFly [2] was among the A-CDN pioneers, followed then by other companies including Edgecast and CloudFlare to name the most popular ones. My own recent work [73] shows that A-CDN technology is mature and readily available, with Internet service providers (e.g. AT&T Services), social networks (e.g. Twitter) and cloud providers (e.g Microsoft) having adopted IP anycast to provide stateful services.

However, to the best of my knowledge, previous works that focused on A-CDNs limitedly leveraged *active measurements* to discover geolocation of anycast replicas, or to benchmark the performance of some specific deployment (see Sec.4.2 for more details). These works show considerable interest in the topic, but they fall short in providing an actual characterization of A-CDNs from the end-user point of view: in other words, how much traffic do they serve? which services do they offer? etc. In order to answer these questions, I leverage *passive measurements*, and offer a characterization of traffic served by A-CDNs in real networks. I use the IP anycast prefixes discovered by the largest census in [73], where about 1600 /24 subnets have been discovered hosting IP anycast servers. This list is compiled using active probing. Given this list, with the aim at providing a first characterization of modern usage of A-CDNs, I use traffic traces from approximately 20,000 households collected from a large European ISP for the entire month of March 2015 (when the census was performed). This large dataset allows us to obtain a snapshot of (i) how popular A-CDNs are, (ii) which services they support, (iii) which are the characteristics of traffic they serve and (iv) their proximity and affinity performance.

[2] http://www.cachefly.com/about.html

I summarize my main findings as follows:

- I confirm IP anycast to be not anymore relegated to the support of connectionless services: in a month I observe over 16,000 active anycast servers contacted via TCP, mapping to more than 92,000 hostnames.
- While hard to gauge via passive measurement, and despite the limited scope that my single vantage point offers, in general I observe stable paths, with only a handful changes during one month.
- Both large players like Edgecast or Cloudflare, and smaller but specialized A-CDNs are present: content served include heterogeneous services such as Twitter Vine, Wordpress blogs, TLS certificate validation and BitTorrent trackers. Footprint of A-CDN is also very different, with some being pervasive enough to have servers at few ms from customers, while others have fewer replica nodes that turn out to be more than 100ms far away.
- In my datasets, A-CDNs are fairly popular: 50% of users encounter at least an A-CDN server during normal web activity. Thus, penetration of A-CDN is already very relevant.
- Most of TCP connections last few tens of seconds and carry a relatively small amount of bytes; surprisingly however, I see video and audio streaming services being supported by A-CDNs, whose TCP flow last for several hours. The latter could be affected by sudden routing changes that could break TCP connections. However, given the infrequent occurrence of such events, it is hard to measure (and suffer from) it in practice.

4.2 Related work

I can categorize the large body of work that investigates IP anycast in three coarse categories: (i) anycast improvement proposals, (ii) studies of anycast performance of specific deployments, (iii) broad assessment of anycast adoption. As for (i), several studies, starting from seminal work such as [74] and culminated recently in [72], propose architectural improvements to address the performance shortcomings of IP anycast in terms of scalability and server selection. They differ from my work since I instead aim at studying the broad variety of existing deployments in the wild.

Concerning (ii), several works focus on deeply studying specific aspects of anycast performance. With few exceptions [75], and in spite of evidence [76] that anycast is successfully used beyond the DNS realm, most of the work targets DNS as the anycast service of interest. The typical key performance indicators include server proximity [77–80, 75], client-server affinity [81, 77–79, 82, 75], server availability [78, 79, 83], and load-balancing [79, 82]. All the methodologies involve active measurements, and thus they are orthogonal to this work. Closest work to ours under this perspective are [82, 77], which base their investigations on passive measurement methodologies. In both cases, authors' collection point is located at the anycast servers, so that they obtain a complete view of all user requests served by specific single server. Conversely, my vantage point is located close to the customers in an ISP network, and it allows us to gather a complementary view of *all anycast services*, albeit from a possibly limited set of users and a single country.

Finally, concerning (iii) there has been a renewed interest in a broad assessment of anycast services, with techniques capable of detecting anycast usage [84], or even enumerate [85] and geolocate [86, 73] anycast replicas. All these techniques perform active measurement to infer some properties of anycast services: in particular, [84, 73] perform censuses of the IPv4 address space to find evidence of anycast deployments via active measurement on the data plane. Despite these studies provide a broad characterization of anycast deployment, they however fail in capturing how popular such services are, how much traffic they attract, and which applications they serve. These are precisely the questions I address in this chapter, taking the census of IP anycast subnet discovered in [73] as a starting point, and using a passive methodology to complement and refine the general picture that can be gathered with active measurements.

4.3 Methodology

4.3.1 Anycast subnet lists

In a nutshell, my workflow first extract the subset of flows that are directed to anycast servers, and then characterize the traffic they exchange with actual internet users by leveraging passive measurements. To identify anycast servers, I rely on the *exhaustive* list of /24 subnet prefixes that result to host at least one anycast server

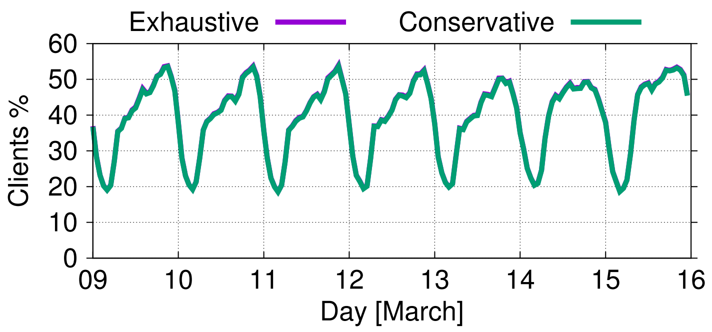


Fig. 4.1 Percentage of clients that contact at least one A-CDN server in each 1h time slot, both curves overlap.

according to the IP censuses I performed in March 2015 [73].[3] As described in [73], I compile an *exhaustive* list of 1696 anycast subnets, by simultaneously pinging an IP/32 from all valid IP/24 networks from PlanetLab probes. The scan runs on all IP address range. Next, I ran the anycast detection technique developed in [86] to identify IP/32 anycast addresses (i.e., located in more than one geographical location). Intuitively, I look for RTT measurements that violates the propagation time constraint from different vantage points. For instance, when pinging an host from two probes, the sum of the RTT measurements to the same server IP must be higher than the propagation time from the two probes. I flag as anycast network any network having at least two locations in my censuses. From this list, I extract a more *conservative* set of 897 subnets, having at least five distinct locations. Since the conservative set is biased toward larger and likely more popular deployments, I expect the conservative set to yield an incomplete but comprehensive picture of IP anycast. In the following, I use this list to inform the passive monitor about the subnets of interest.

4.3.2 Passive monitor

To collect my dataset, I instrumented a passive probe at one PoP[4] of an operational network in an European country-wide ISP with Tstat as explained in Sec. 1.1.

[3]Recall that BGP announced prefixes have a minimum granularity of a /24 subnet. Thus, an anycast address range cannot be smaller than a /24 range.

[4]Results from other vantage points in other PoPs are practically identical. For easy of presentation, I focus on one PoP in this work.

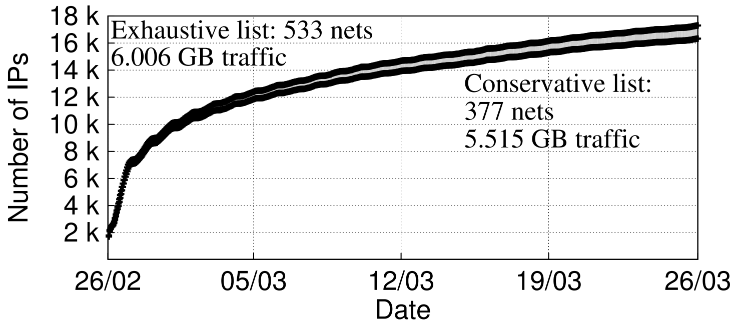


Fig. 4.2 Cumulative number of distinct servers encountered over the month.

In particularity, for this study I leverage a dataset of TCP flows collected during the whole month of March 2015. It consists of 2 billions of TCP flows being monitored, for a total of 270 TB of network traffic. 1.4 billion connections are due to web (HTTP or HTTPS) generating 209 TB of data. More important, I observe more than 20,000 ISP customers active over the month, which I identify via the static anonymized client IP address[5]. All traffic generated by any device that accesses the internet via the home gateway is thus labeled by the same client IP address. This includes PCs, smartphone, Tablets, connected TV, etc. that are connected via WiFi or Ethernet LAN to the home gateway.

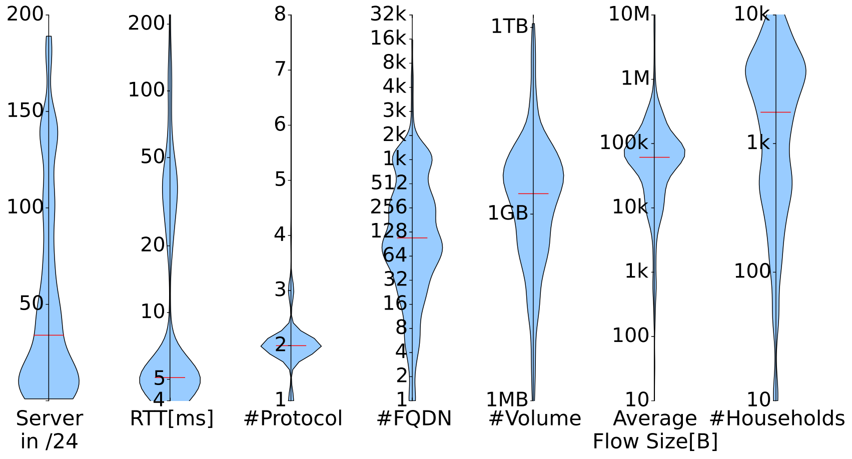
Among the many measurements provided by Tstat for each TCP flow, I focus only on: (i) the server IP address; and (ii) the anonymized client IP address; (iii) The minimum Round-Trip-Time (RTT) between the Tstat probe and the server; (iv) The amount of downloaded bytes; (v) The application layer protocol (e.g., HTTP, HTTPS, etc.); (vi) The FQDN of the server the client is contacting as returned by DN-Hunter.

4.4 Anycast at a glance

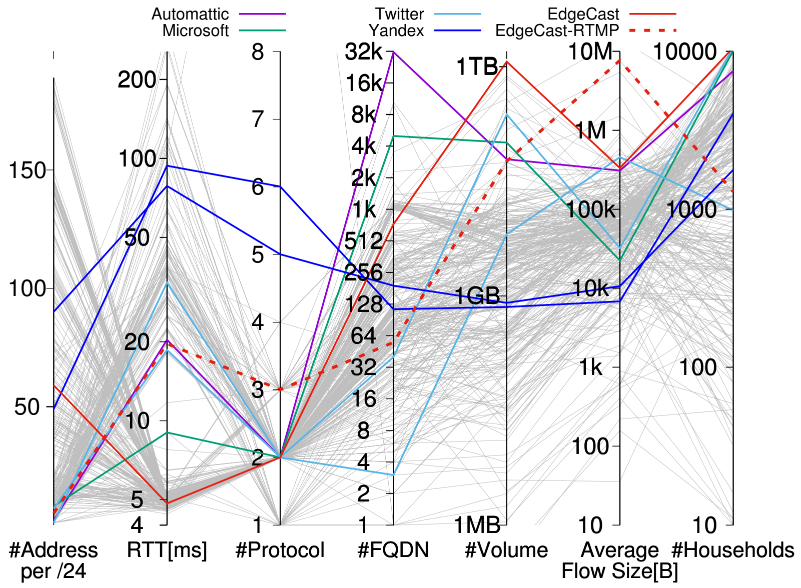
4.4.1 Temporal properties

I first provide an overall characterization of the anycast traffic. Not surprisingly, I observe that all anycast UDP traffic is labeled as DNS protocol – which I avoid

[5]The ISP adopts a static addresses allocation policy, so that each customer home gateway is uniquely assigned the same static IP address.



(a) violin plots



(b) Parallel coordinate plot. For the ease of readability, single curve plots can be found at [87]

Fig. 4.3 Anycast at a glance[6]

investigating given the literature on anycast DNS. More interestingly, I observe a sizeable amount of anycast traffic carried over TCP: overall, almost 59 million TCP connections are managed by anycast servers. Those correspond to approximately 3% of all web connections and 4% of the aggregate HTTP and HTTPS volume, for a total of 6 TB of data in the entire month. Definitively a not-negligible amount of traffic, especially when considering the relatively small number of /24 anycast subnets.

The large majority of traffic is directed to TCP port 80 or 443, that the DPI classifier labels as HTTP and SSL/TLS, respectively. This suggests that hosted services are indeed offered by A-CDNs and served over HTTP/HTTPS. A minority of the traffic (less than 1% of all anycast traffic) is instead related to some protocols for multimedia streaming, email protocols, Peer-to-Peer traffic, or DNS over TCP. I will provide further details when digging into some selected examples.

Results confirm the footprint of anycast traffic, and A-CDN in particular. To corroborate this, Fig. 4.1 shows evolution during one-week of the percentage of active customers that have encountered at least one anycast server during their normal web browsing activities (the ratio is computed at hourly intervals, normalizing over the number of client active in that hour). Besides exhibiting the day/night pattern due to the different nature of services running on the network with fewer services served over anycast at night, the figure shows that at peak time the probability to contact at least one anycast server is higher than 50%. Notice that Fig. 4.1 reports the probabilities according to both the exhaustive and the conservative lists: these curves cannot be distinguished as they perfectly overlap, which hints to the fact that the conservative list is, as expected, comprehensive enough for my purposes. This clearly may change on vantage points located in different countries. However, for the purpose of this work, I prefer to take a conservative approach.

Fig. 4.2 reports the cumulative number of unique IP anycast addresses observed over time, again for both the conservative and exhaustive lists. In total, over 16,000 distinct IP addresses are observed during the whole month for the conservative list. The picture is additionally annotated with the traffic volume exchanged with such servers: notice that despite the exhaustive list is twice as big as the conservative one, the number of servers contacted and bytes exchanged are fairly similar. This happens since the major A-CDN players are present in the conservative list, to which I thus limitedly focus on the following. Notice also that the number of distinct servers encountered over the month quickly grows during the first days, during which most popular services and servers are contacted.

/24 subnet	Owner	IP/32	Vol. [GB]	Flows [k]	Users	FQDN	Protocols	Content/Service
93.184.220.0	EdgeCast-1	105	357	8,018	10,626	3,611	HTTP/s	generic
199.96.57.0	**Twitter-generic**	7	219	7,318	10,508	40	HTTP/s	twitter, vine
68.232.34.0	**EdgeCast-2**	59	1,071	3,484	10,490	736	HTTP/s	microsoft, spotify
68.232.35.0	EdgeCast-3	104	480	5,059	10,354	904	HTTP/s	twitter, gravatar, tumblr
94.31.29.0	NetDNA	73	80	1,292	10,218	609	HTTP/s	generic
93.184.221.0	EdgeCast-4	49	708	2,031	10,155	1,467	HTTP/s	generic
204.79.197.0	**Microsoft**	8	93	4,508	10,044	5,088	HTTP/s	bing, live, microsoft
205.185.216.0	Highwinds-1	2	180	1,411	9,705	267	HTTP/s	generic
108.162.232.0	CloudFlare-1	13	53	550	9,274	14	HTTP	ocsp certificates
178.255.83.0	Comodo	5	3	601	8,837	65	HTTP	ocsp certificates
192.0.72.0	**Automattic**	2	57	199	7,477	32,037	HTTP/s	wordpress
108.162.206.0	CloudFlare-2	122	14	246	4,695	465	HTTP/s, Torrent	P2P trackers, generic
213.180.193.0	**Yandex-2**	49	0.7	105	4,031	114	HTTP/s SMTP	yandex
213.180.204.0	**Yandex-1**	90	0.7	76	1,771	191	HTTP/s, SMTP	yandex
93.184.222.0	**EdgeCast-RTMP**	5	53	8	1,289	55	RTMP, HTTP	soundcloud, video
199.96.60.0	**Twitter-vine**	1	6	14	983	3	HTTP/s	vine
Total	Exhaustive	17,298	6,006	58,885	10,830	120,151	-	-
Total	Conservative	16,329	5,515	54,045	10,828	117,768	-	-

Table 4.1 Dataset summary

4.4.2 Service diversity

I now provide an overall picture of A-CDN diversity with a dual violin plots (top) and parallel coordinate (bottom) representation in Fig.4.3. Violin plots compactly represent the marginals for some metrics of interest, whereas parallel coordinate plots allow us to grasp the correlation between these metrics for some specific deployments. On both plots, I select the following axes: (i) the number of active servers in the /24 subnet, (ii) the average minimum RTT for any server in that /24, (iii) the number of distinct protocols, (iv) the number of distinct FQDNs/services, (v) the total amount of bytes served during the whole period, (vi) the average flow size in bytes, and (vii) the number of households that contacted one server in the /24.

In more details, violin plots of Fig.4.3 are an intuitive representation of the Probability Density Function (PDF): the larger their waist is, the higher is the probability of observing that value; red bars are a further visual reference, corresponding to the median of the distribution. Overall, the plot shows that most of /24 host few servers, which are in general quite close to the PoP (RTT<10 ms), use 2 or at most 3 protocols (HTTP or HTTPS mostly). Diversity starts to appear in the number of served FQDNs – with some /24 being used for a handful services, while others serve several thousands. Served volume varies widely. Similarly, while only half of flows exceed 50 kB, and most are below 1 MB, flow size peaks up to several hundreds MB (see Sec.4.3 for more details). Considering popularity, some /24 are used by several thousands end-users, others by less than 10.

Parallel coordinates instead allow the observation of a specific deployment: each line represent a /24 subnet, and the "path" among the vertical axes highlights the characteristic of that A-CDN over different dimensions. I report most[7] /24 with light gray color, and additionally I highlight some of them using different colors. First, observe that the wide dispersion of the light gray paths testifies great diversity across A-CDN deployments.

Next, observe per-deployment dispersion of the few selected /24. For the sake of illustration, consider the Automattic curve, which is the A-CDN that serves websites hosted by Wordpress: it can be seen that the two active servers found in the /24 are located at about 20 ms from the PoP. They use both HTTP and HTTPS, for a total of more than 32,000 FQDNs. Total volume accounts for 20 GB during the month, transferred over flows that are 200 kB long on average. At last 7400 users (37%) accesses some content hosted by Automattic. Without going into details for lack of space, I have highlighted telling examples such as: Twitter A-CDN (which serve few domains); Microsoft A-CDN (bing.com and live.com services, for a total of more than 5000 FQDNs); one /24 of EdgeCast as an example of a generic A-CDN; a specialized EdgeCast platform serving audio and video streams over Real Time Media Protocol (RTMP), see the red dashed line. Finally, I selected two /24 belonging to Yandex, the most popular search engine and social platform in Russia, that are however not among the most popular in the geographic region of my vantage point. It clearly appears that these examples, which I more deeply investigate in the rest of the chapter, are representative of quite diverse anycast deployments. This makes it difficult to highlight common trends, e.g., I observe popular A-CDN whose RTT is quite large, and unpopular A-CDN whose RTT is instead much smaller.

4.5 Selected anycast deployments

4.5.1 Candidate selection

Tab. 4.1 offers details for some selected deployments. Rows highlighted in bold font refers to the same subnets previously highlighted in Fig. 4.3.

[7] To reduce visual cluttering, I report in light gray color the subset of /24 that served at least 1000 flows and 10 distinct households during a month.

For each /24 subnet, Tab. 4.1 lists the Owner (i.e., the organization managing it as returned by Whois), the number of distinct server addresses that have been contacted at least once, the total volume of bytes served, the number of flows, of users, and of distinct FQDNs. The last two columns report the most prominent protocols and services the A-CDN offers.

The table, which also serves as a summary of my anycast dataset, comprises the top-10 most popular /24 A-CDN prefixes and subnets (top part). To avoid an excessive bias toward only popular services (where HTTP and HTTPS are predominant), I additionally report some manually selected A-CDNs (bottom part).

As it can be observed, the portfolio of services supported by A-CDNs includes email via SMTP, video/audio streaming via RTMP, certificate validation via Online Certificate Status Protocol (OCSP), and even BitTorrent Trackers.

The table precisely quantifies the very heterogeneous scenario early depicted by the Fig. 4.3. EdgeCast is the major player in my dataset, managing 4 of the top-10 subnets: each of these serves between 350 GB/month and 1 TB/month to more than 10,000 (50%) households in my PoP. This is not surprising, given EdgeCast claims to serve over 4% of the global Internet traffic[8].

Popular A-CDNs includes Microsoft, which directly manages its own A-CDN. It serves Bing, Live, MSN, and other Microsoft.com services. Since it handles quite a small amount of data and flows, I checked if there are other servers not belonging to the Microsoft A-CDN that handle those popular Microsoft service. I found that all of *bing.com* pages and web searches are actually served by the Microsoft A-CDN, while static content such as pictures or map tiles are instead by the Akamai CDN. Thus, Microsoft is using an hybrid solution based on a traditional CDN and its own A-CDN at the same time.

Finally, popular A-CDN services include Highwinds and Comodo. Highwinds offers video streaming for advertisement companies, and images for popular adult content websites (notice the relative longer lived content). Instead, Comodo focuses its business on serving certificate validations using OCSP, Online Certificate Status Protocol, services (with lot of customers who fetch little information).

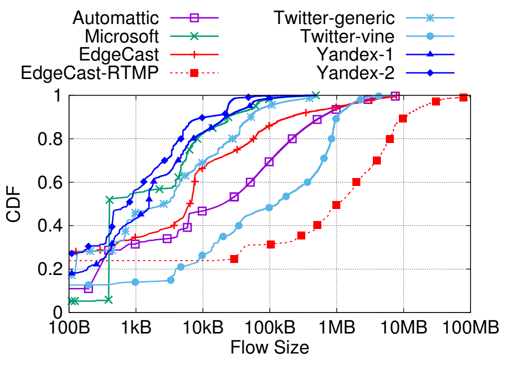
Overall, major A-CDN players serve thousands of FQDNs including very popular web services like Wordpress, Twitter, Gravatar, Tumblr, Tripadvisor, Spotify, etc.

[8]http://www.edgecast.com

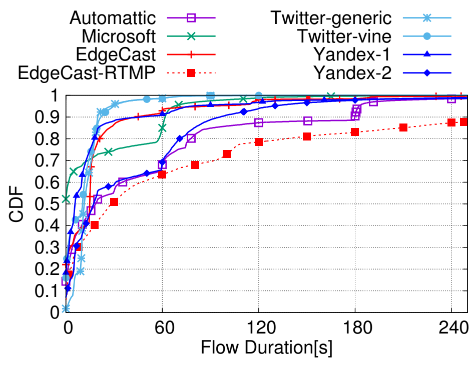
This explains why about 1 out of 2 end-users likely contacts at least one A-CDN server during her navigation. Most FQDNs are uniquely resolved to one IP address – but the same IP address serves multiple FQDNs. Interestingly, this behaviour is shared among most of the studied A-CDNs, meaning that they purely rely on anycast routing for load-balancing. An exception is CloudFlare's A-CDN which uses also DNS load-balancing. Cloudflare uses up to 8 IP addresses in the same /24 to serve the same FQDN. For lack of space, I am not able to report here the results of a complementary active measurement study that I report at [87].

4.5.2 Per-deployment view

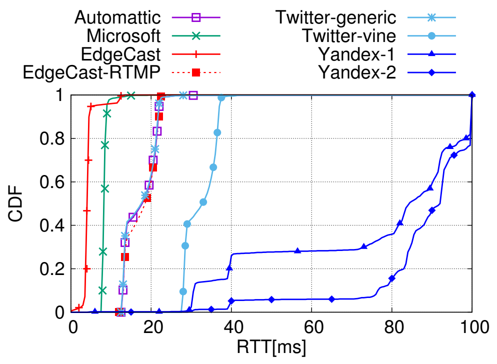
For the selected deployments, I details the cumulative distribution function (CDF) of some interesting metrics. The CDF is computed considering all the flows being served by the same /24 subnet.



(a) Flow size CDF



(b) Flow duration CDF



(c) Round-trip time CDF

Fig. 4.4 Metrics characterization

As for the metrics of interest, I report the CDF of flow size (Fig. 4.4(a)), duration (Fig. 4.4(b)), and Round Trip Time (Fig. 4.4(c)). Fig. 4.4(a) shows how the size of the content hosted by A-CDNs varies across deployments (the different supports only partly overlap), and also between flows of the same deployment (the support is large and, with few exceptions, there is no typical object size). In general served objects are shorter than 1 MB, with the notable exception of audio and video streams served by EdgeCast specialized deployment that support RTMP streaming. In this case, flows are larger than 100 MB.

The small amount of data is reflected on the TCP flow duration CDF. Indeed, Fig.4.4(b) shows that flow lifetime is in general shorter than 180 s, with specific values that reflects typical HTTP server timeouts (multiple of 60 s). Once again, the only exception is the EdgeCast-RTMP deployment, for which over 10% of the TCP flows exceed 5 minutes (visible in the picture), and ranges up to hours (not visible in the picture). Finally, minimum Round Trip Time CDF in Fig.4.4(c) reveals that the popular A-CDNs have a good footprint (at least in Europe). The only exception is Yandex that have anycast replicas in the eastern Europe and in Russia (which is not surprising due to the language specific content it serves).

Notice how sharp the CDF are for EdgeCast or Microsoft deployment. This suggests that the path to their servers is very short, but also very stable. RTT of other A-CDNs is instead very similar, e.g., EdgeCast-RTMP, Twitter-generic, and Automattic. This suggests that their servers are located in the same place, and reached by the same path. Interestingly, the minimum RTT shows a deviation which suggests the presence of some extra delay for 60% of samples, possibly accounting for some queuing delay due a possibly congested link on that path (which belongs to the Twitter-vine path as well).

4.5.3 RTT variation over time

In this section I look for evidences may suggest possible path properties changes. I based my analysis studying the TCP minimum RTT. Intuitively, a *sudden* change in the minimum RTT could highlight a possible sudden change in the routing or network infrastructure. I argue that a dramatic change in the TCP minimum RTT, is likely due to a routing change. Conversely, a *smooth* shift could suggest the presence of queuing delay due to possible congestion on some path links. Passive measurements

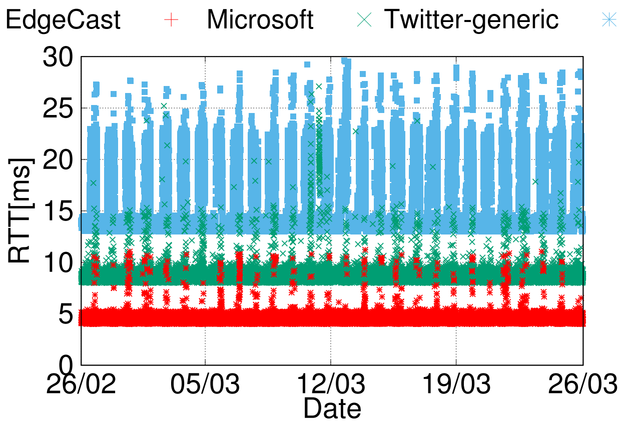
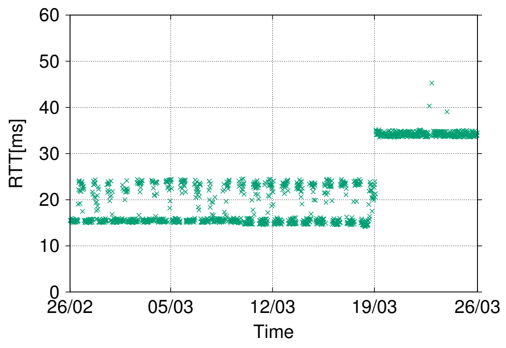
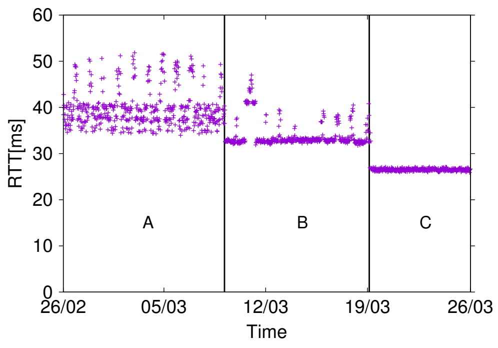


Fig. 4.5 Stable Situation. Single curve plots can be found at [87]



(a) Event 1: sudden change



(b) Event 2: multiple changes

Fig. 4.6 RTT changes

could only suggest to investigate more deeply of eventual changes, e.g., triggering active measurements to provide a more reliable ground truth to distinguish between hardware improvement and routing changes. For instance, checking HTTP headers could be used to reliably reveal the anycast replicas [73].

Fig. 4.5 shows the minimum RTT values for each TCP flows over time. Three most used A-CDNs during the entire month of March 2015 are considered. The figure suggests that no sudden changes in the path are visible. Yet, observe the periodic and smooth increase of RTT during the peak time. This could be explained as a congestion events, that are reflected in the smooth changes in the minimum RTT CDF as shown in Fig. 4.4(c). For these three A-CDN, data suggest stable but possibly congested paths to the anycast addresses.

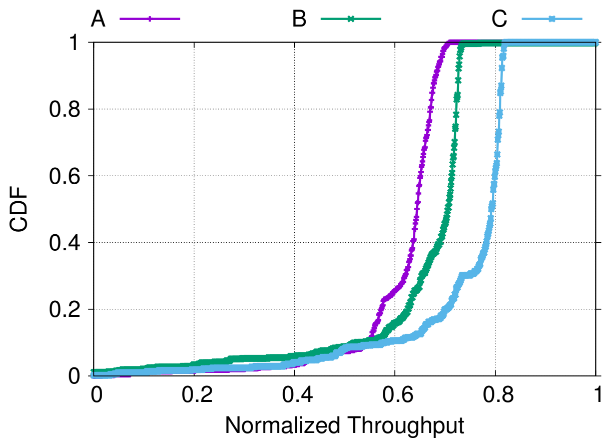


Fig. 4.7 Event 2: Throughput Implications

I investigated the RTT evolution over time of other /24 subnets. I found few cases that I believe could highlights possible sudden changes in the routing plane, possibly affecting server affinity. Fig. 4.6(a) and Fig. 4.6(b) report two examples. Plot on the top shows an example of sudden changes affecting another /24 subnet. In this case, the minimum RTT suddenly increases by almost 15ms. Notice also that the path to the longer location is not affected by periodical increases during peak time, suggesting no congestion is present in this second path. Look now at the plot on the bottom. It suggests changes on March 9th, and 19th (with possibly a short change on the 11th). Indeed, minimum RTT properties differ quite significantly and with a sharp change. To study the implication of this from a client perspective, I report the CDF of the throughput for the three distinct periods in Fig. 4.7. I normalize the throughput between 0 and 1 for ISP privacy motivation. The three distribution show that when the RTT decreases, i.e., the server serving the flow is closed to the users, the throughput improves. Thus, A-CDN changes have an impact on performance as well. I have observed other changes in different /24 networks, not reported here due to lack of space.

I also tried to investigate if changes have implications on TCP connections. In particular, one would expect that an on-going TCP connections to be abruptly terminated if the routing change implies a server change as well. I tried to investigate this by correlating number of TCP flows abruptly terminated by a server RST message with possible routing change events. I am not able to observe any clear evidence. Indeed, on the one hand, TCP flows are very short – cfr. Fig. 4.4(b) – and, on the other hand, changes are sudden and very few. Thus only an handful TCP connections could possibly be involved during a change event. This support the intuition that anycast is indeed well suited for connection oriented services.

In summary, while I observe sharp changes in the anycast path to reach the A-CDN caches, those events are few and occasional, with each different routing configuration that lasts for days. Clearly, deeper investigation is needed to better understand eventual routing changes over the time. In this direction I am trying to exploit other metrics as the Time To Live (TTL) and the Time To First Byte (TTFB) to highlight routing changes. Combining this methodology with active measurements, it could provide a better understanding of routing stability that may affect A-CDN deployments.

4.6 Conclusions and Discussion

I presented in this chapter a first characterisation of Anycast-enabled CDN. Starting from a census of anycast subnets, I analysed passive measurements collected from an actual network to observe the usage and the stability of the service offered by A-CDNs. My finding unveil that A-CDNs are a reality, with several players adopting anycast for load balancing, and with users that access service they offer on a daily basis. Interestingly, passive measurements reveal anycast to be very stable, with stable paths and cache affinity properties. In summary, anycast is increasingly used, A-CDNs are prosperous and technically viable.

This work is far from yielding a complete picture, and it rather raises a number of interesting questions such as:

Horizontal comparison with IP unicast. Albeit very challenging, more efforts should be dedicated to compare Unicast vs Anycast CDNs for modern web services. To the very least, a statistical characterization and comparison of the pervasiveness of the deployments (e.g., in term of RTT) and its impact on objective measures (e.g., time to the first byte, average throughput, etc.) could be attempted.

Vertical investigation of CDN strategies. From my initial investigation, I noticed radically different strategies, with e.g., hybrid DNS resolution of few anycast IP addresses, use of many DNS names mapping to few anycast IPs, use of few names mapping to more than one anycast IPs, etc. Gathering a more thorough understanding of load balancing in these new settings is a stimulant intellectual exercise which is not uncommon in my community.

Further active/passive measurement integration. As anycast replicas are subject to BGP convergence, a long-standing myth is that it would forbid use of anycast for connection-oriented services relying on TCP. Given my results, this myth seems no longer holding. Yet, while I did not notice in my time frame significant changes in terms of IP-level path length, more valuable information would be needed from heterogeneous sources, and by combining active and passive measurements.

Chapter 5

SeLINA: a Self-Learning Insightful Network Analyzer

5.1 Introduction

This chapter presents *SeLINA: a Self-Learning Insightful Network Analyzer* a paper published in the journal *IEEE Transactions on Network and Service Management.*[1]

The work I present in this chapter has been developed with the DataBase an Data Mining Group (DBDMG) at the Politecnico di Torino. I joined this group as a network expert to helped them on the validation of the SeLINA tool. In particular, I offered my skill to verify if SeLINA is able to highlight network traffic anomalies. My contributions will be presented in Sec. 5.7 related to the study of the YouTube anomaly previously highlighted in Sec. 2.6.1, and in the analysis of peer-to-peer traffic in Sec. 5.8.

Internet monitoring has always played a fundamental role in understanding how the network is performing, how users are accessing resources, and how to properly control and manage the infrastructure. The growth of traffic, users, services and applications running in the Internet challenges everyday the network managers and analysts to cope with system complexity and in the understanding of how it

[1]Daniele Apiletti, Elena Baralis, Tania Cercquitelli, Paolo Garza, Danilo Giordano, Marco Mellia, and Luca Venturini, "SeLINA: a Self-Learning Insightful Network Analyzer" in IEEE Transactions on Network and Service Management, vol. 13, no. 3, pp. 696-710, August 2016. doi: 10.1109/TNSM.2016.2597443

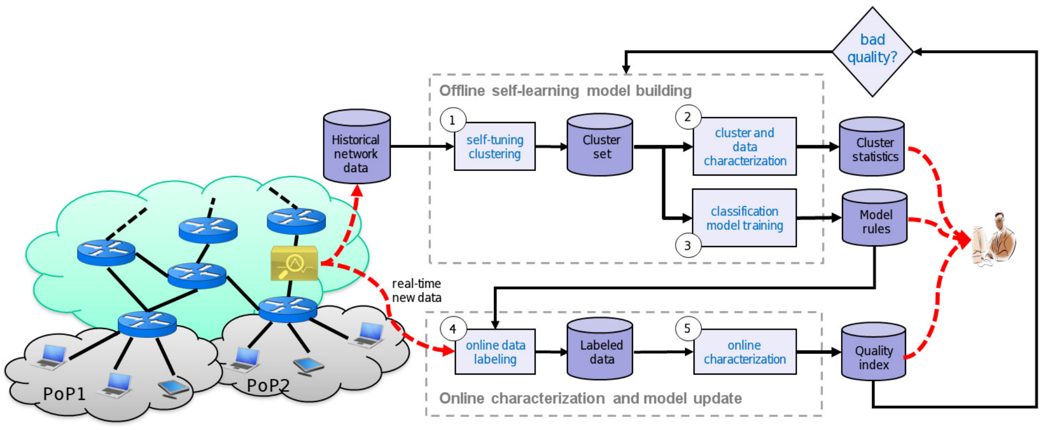


Fig. 5.1 SeLINA building blocks.

works. big data and machine learning approaches have emerged to build systems that aim at automatically extracting information from the raw data that the monitoring infrastructures offer, and a significant effort has been devoted to apply them to network traffic analysis. Most of the proposed systems target a specific problem, e.g., monitoring of a CDN [3, 4, 88], detecting anomalies [5, 6], or simply offering scalable platforms [89].

However, few works have targeted the general-purpose extraction of useful information from the raw data exposed by the system, i.e., the application of the data mining approach to information discovery, a classic application of unsupervised machine learning approaches. While methodologies exist, to the best of our knowledge, they require non-trivial skills and the domain expert needs to be able to fine tune the underlying algorithms. In this work, we target the design of an unsupervised machine learning tool that allows the network administrator to discover properties of the traffic, without requiring her to be a machine learning expert. we identified the following requirements.

- Scalability, as the ability to (i) process very large datasets, but (ii) provide compact representations of the traffic, independently of the data size.
- Auto-configuration, as the capability to (i) self-adapt to different data distributions (e.g., data densities, cluster shapes), and to (ii) self-tune the algorithm parameters to avoid human intervention.
- Human-readability of both results and underlying models, to make the knowledge better exploitable and more actionable.

- Self-assessment and self-evolution, to autonomously evaluate the model quality and trigger a rebuilding when the model fitting to new data degrades.

The above-mentioned design guidelines led to the design of SeLINA (Self-Learning Insightful Network Analyzer), which exploits both supervised and unsupervised data-mining techniques by combining their strengths. Specifically, effective unsupervised approaches are used to autonomously identify clusters of homogeneous traffic flows, thus reducing the granularity of objects to observe from millions of single flows to few tens of clusters, and generating a model of traffic. Human-readable and fast supervised approaches are used then to classify flows on the fly and assign them to clusters, and to offer valuable information about the main characteristics of each class. The system computes internal quality indices to check whether the new data does not fit anymore the historical model, suggesting to the analyst changes in the new network traffic, and, possibly automatically, triggering a new clustering phase to update the model.

SeLINA has been implemented in a state-of-the-art big data framework, Apache Spark, and has been applied to two real-world large use cases: a YouTube video streaming dataset and a peer-to-peer traffic dataset. Experimental results showed that SeLINA is able to provide insightful network traffic models, e.g., pinpoint different groups of YouTube servers with different properties, and suggest the presence of changes in the infrastructure that have caused well-known issues to end-users [88].

This chapter is organized as follows. Section 5.3 provides an overview of the proposed methodology, while Sections 5.4 and 5.5 describe its main building blocks. Section 5.6 provides and overview of the experimental evaluation campaign, while Sections 5.7 and 5.8 thoroughly discuss the experiments performed on two real use cases based on real traffic datasets. Finally, Section 5.2 compares our approach with previous work, while Section 5.9 draws conclusions and presents future developments of this work.

5.2 Related work

A significant effort has been devoted to the application of data mining techniques and statistical methods to network traffic analysis. The application domains include studying correlations among network data (e.g., association rule extraction for network traffic characterization [90, 91]; for router misconfiguration detection [92]; interesting correlations from web-based e-business system [93]), extracting information for prediction (e.g., multilevel traffic classification [94], Naive Bayes classification [95], throughput prediction [96], analytics and statistical models for LTE Network Performance [11], one-class SVM [97] for intrusion detection), grouping network data with similar properties (e.g., clustering algorithms for intrusion detection [98–100, 5, 6], for deriving node topological information [101], for automatically identifying classes of traffic [102–105], for unveiling YouTube CDN changes [88]), and context specific applications (e.g., multi-level association rules in spatial databases [106]).

However, in most cases no approach offloads the user from arbitrary parameter choices, and can be easily adapted to domain-specific requirements and semantics as the methodology proposed in this chapter. Differently from analytics approaches tailored to a specific network application [88, 98, 99, 101, 100, 5, 6], SeLINA is a general purpose methodology that can be easily exploited to analyze different and transversal network data (e.g., network traffic headers, network flow characteristics, statistical measurements of traffic flows). In the experimental section we considered Youlighter [88]. Youlighter is a system that detects very specific macro-changes in the YouTube traffic pattern involving the CDN spatial distribution. It is not distributed nor scalable. SeLINA, instead, introduces a general-purpose, distributed, and fully autonomous engine exploiting a completely different methodology and addressing a more general research issue in the network traffic analysis than the Youlighter system.

The performance of most state-of-the-art general purpose approaches [90, 91, 102–105] depends on the choice of different parameters, and the optimal trade-off between execution time and accuracy must be handpicked for a given application. On the contrary, focusing itself on self-learning capabilities of state-of-the-art scalable approaches, SeLINA is able to build a model of the data with minimal user intervention by offloading the user from the non trivial task of configuring the miner system and highlighting possibly meaningful interpretations to domain experts.

Some research effort has been devoted to automatic setting of data mining algorithm parameters (e.g., clustering algorithms [107, 108], itemset mining [109]). Authors in [107, 108] proposed a hierarchical strategy to aggregate lower density regions discovered through DBSCAN. Different from [107, 108], SMDBSCAN automatically sets DBSCAN parameters at each iteration level when DBSCAN is exploited in a multiple-level fashion. Furthermore, the SeLINA clustering results include clusters with a diverse degree of density, because each subset of clusters with a similar density is discovered at a given iteration level. The method in [107, 108] instead gets a flat partition composed of clusters extracted from local cuts through the cluster tree.

An intensive research activity has been devoted to designing innovative algorithms and methodologies to support large scale analytics based on MapReduce, such as [110–112]. A step further has been proposed in [113]. Apache Spark with its Resilient Distributed Datasets and its smart APIs, outperforms Hadoop MapReduce in terms of performance and overcomes its limitations, with particular focus on iterative in-memory computation, which is a common characteristic of many data mining algorithms. Its machine learning library MLlib [114] provides a broad range of analytics algorithms. SeLINA exploits the computational advantages of distributed computing frameworks, as the current implementation runs on Spark. Applications of this techniques to network traffic analysis becomes natural, given the volume of traffic [89, 115–117]. These works adopt Hadoop or Spark, and apply either standard machine learning algorithms, or design specific solutions to their problem.

The idea of defining a generic framework and of tightly integrating self-learning capability in a scalable data mining engine tailored to traffic data was first introduced by ourselves in [118]. However, SeLINA significantly enhances the methodology proposed in [118]. The SeLINA data mining engine (named SaFe-Nec in [118]) provides an innovative and more accurate explorative approach coupled with self-configuring strategies (i.e., the SMDBScan algorithm). Thus, SeLINA allows exploiting cluster analysis on real datasets in a fully autonomous fashion. The SMDBScan algorithm is characterized by configuration parameters whose setting is rather difficult. In [118] the less effective, but easier to configure, K-means algorithm was used. SeLINA also includes ad-hoc strategies to automatically tune the clustering parameter values, which is a typically difficult task also for domain experts. The exploitation of a multilevel DBSCAN-like algorithm jointly with self-configuring strategies allowed for better clustering results than the ones produced by the K-means based approach proposed in [118]. Moreover, SeLINA integrates innovative self-assessment features and a new set of network domain statistics that are often vital to let the domain expert interpret the results. Finally, with respect to [118], in this work we added a new interesting case study on YouTube traffic analysis and a thorough analysis of the results from a networking point of view.

5.3 Methodology Overview

Figure 5.1 depicts the main components of the proposed methodology.

5.3.1 Offline self-learning model building.

This component, which analyzes historical network traffic flows, aims at building a self-learning data characterization model, and consists of three phases: (1) a self-tuning clustering phase, (2) a cluster and data characterization phase, and (3) a classification model training phase.

5.3.2 Online characterization and model update.

This component analyzes new network data in real-time by applying the model built in the previous block to detect changes in the network traffic characterization. It consists of two phases: (4) a real-time data labeling phase, and (5) an online characterization phase.

In details, step (1) of the proposed methodology consists of a self-tuning clustering algorithm, which is run over historical data to discover homogeneous groups of traffic flows without prior knowledge, in a fully autonomous and unsupervised fashion. Effectively applying cluster analysis on real datasets requires the non-trivial choice of algorithm-specific parameters, a typically difficult task for domain experts exploiting data mining techniques. To this aim, SeLINA includes ad-hoc strategies to automatically tune the clustering parameter values. In step (2), the resulting cluster set is then enriched by both general-purpose and domain-specific statistics, whose aim is to support network analysts in understanding the semantics of the identified clusters.

The cluster set is also the input to the model training phase of step (3), where a classification model is built by exploiting clusters as classification labels. The model is able to self-learn how to assign each network flow to the proper cluster. Different classification techniques could be exploited, depending on the preference towards pure performance (e.g., accuracy) or human-readability of the model. SeLINA choice is a decision tree algorithm, which is among the most popular classification techniques and provides an easily readable model in the form of classification rules.

The latter feature supports network analysts in getting more meaningful insights on the reasons for the classifier underlying choices.

The classification model is exploited in the real-time data labeling phase at step (4), where each new network flow is assigned a label. Then, at step (5) the quality index computation is executed, by exploiting different quality indicators to self-assess the model fitting and its results over time. When the quality index falls below a given threshold, the offline model building can be automatically triggered to rebuild a model better fitting the new data, thus providing self-evolutionary features.

SeLINA is a general-purpose methodology which can be easily exploited to analyze large collections of network data (e.g., network traffic headers, network flow characteristics, statistical measurements of traffic flows). As a case study, in this work we apply SeLINA to analyze network measurements collected through Tstat.

5.4 Offline self-learning model building

The core of the SeLINA approach is the offline self-learning model building component, which consists of (1) a self-tuning clustering phase, (2) a cluster and data characterization phase, and (3) a classification model training phase. Details on each phase are provided in the following.

5.4.1 Self-Tuning Clustering Phase

SeLINA exploits clustering to autonomously identify homogeneous groups of network traffic flows without prior knowledge. This phase performs a preliminary data normalization step, by means of the standard z-score technique [119], and then a clustering algorithm is applied to the normalized data. Among the many clustering algorithms available to this aim, SeLINA provides an advanced DBSCAN-based [120] algorithm [121] providing high-quality clusters on very-large real data collections. Since the clustering phase is at the core of the SeLINA self-learning feature, in the following subsections its building blocks are presented.

Basic DBSCAN

DBSCAN is the density-based clustering algorithm already presented in Sec. 2.4.1. we applied DBSCAN due to: the ability of identifying arbitrary-shaped clusters, isolate noise and outliers, and because the results provided by DBSCAN are usually better than those provided by other popular clustering algorithms.

However, since DBSCAN requires a long execution times, due to its quadratic complexity, to scale to very large datasets, we exploit the Spark-based distributed implementation of DBSCAN[2] proposed by Aliaksei Litouka. Its main difference with respect to the original centralized version is an additional partitioning step, performed at the beginning.

Self-tuning Multi-level DBSCAN

SeLINA improves the basic DBSCAN approach by addressing two main issues: (i) parameter setting, and (ii) diverse data densities within the same dataset. To offload domain experts from the critical task of configuring DBSCAN-specific parameters, SeLINA includes a self-tuning strategy to automatically set proper values. Furthermore, very-large real datasets are often characterized by diverse data distributions in different regions, and this is an issue hardly handled by the standard DBSCAN version. To address both issues, the SMDBSCAN algorithm in SeLINA builds upon an advanced version of DBSCAN successfully proposed in [121]. SMDBSCAN features a multi-level iterative approach and a smart automatic parameter-setting procedure.

a) *Multi-level iterative approach:*

At each iteration, SMDBSCAN considers the data points which have not been assigned to a cluster yet (at the first iteration, the whole dataset is considered), then (i) partitions them to allow parallel computation, (ii) automatically selects the most appropriate values of $epsilon$ and $MinPoints$, and (iii) executes the standard DBSCAN with such parameter settings. At the end of each iteration, the newly found clusters are included in the global set of clustering results, while the noise points become the dataset for the next iteration.

[2]Downloaded from https://github.com/alitouka/spark_dbscan

Algorithm 1 Epsilon setting

```
Input  : Dataset partitions - $dataPartitions$
Input  : Epsilon step - $epsStep$
Output : Estimate of best epsilon - $bestEpsilon$

List<Double> potentialEpsilons = {}  for $partition$ in $dataPartitions$ do
    for $p$ in $partition$ do
        /* Compute the density of the areas centred in $p$ of a
           radius $eps$ multiple of $epsStep$                              */
        Map<Double,Int> densities = {}  $eps$=$epsStep$;
        $density\_before = 0$;
        $found$ = False;
        while (not $found$ and $eps$ <=distanceFromFarthestPoint($p$,$partition$)) do
            $numNeighbours$ = numNeighboursRadiusEps($p$, $eps$, $partition$)  $density$
            = $numNeighbours/eps^d$ ;  /* $d$ is the number of attributes */
            if ($density < density\_before$) then
                $found$ = True;
            end
            $density\_before = density$
        end
        /* The potential value of $epsilon$ for $p$ is the one before
           the first density decrease                                      */
        $epsilonP = eps$;
        potentialEpsilons.add($epsilonP$)
    end
end
/* Among the potential values of $epsilon$, select the one
   corresponding to the first quartile                                     */
$bestEpsilon$=FirstQuartile($potentialEpsilons$)  return $bestEpsilon$
```

Fig. 5.2 Automatic setting of the *epsilon* parameter value.

b) *Automatic parameter setting: epsilon:*

To automatically compute the values of *epsilon* and *MinPoints* at each iteration, SMDBSCAN introduces a self-tuning procedure, consisting of two heuristics. The two heuristics are very intertwined, the second depending on the *epsilon* set by the first, and they are designed to fit the scope of a multi-level strategy.

To determine *epsilon*, SMDBSCAN exploits the density-based concept of cluster: "a dense area surrounded by a lower density zone". To this aim, a greedy approach is exploited, selecting the best potential *epsilon* for each point separately. A final decision is then taken globally given all the local best *epsilons*.

In particular, given an arbitrary point p, the algorithm tries to identify the boundary of the dense area around p. To this aim, it computes the density distribution in the hypersphere having radius r and center in p, with increasing values of r. The larger r, the higher the number of points inside the hypersphere will be. we define dense areas when the number-of-point increasing rate is higher than the growth in volume. At the border of a dense area, this growth rate will show an inversion of the trend, as soon as the volume starts growing faster than the number of points. The proposed heuristics chooses the first inversion point as the border. If many inversion points occur, greedily choosing the first one reduces the computational time and leaves margin for further exploration in the next levels of SMDBSCAN. The final value of *epsilon*, actually used for each run of DBSCAN, is selected by considering the first quartile of the set of border values generated by applying the border-detection procedure for all points p. The first quartile value produces a set of dense clusters covering a representative subset of our data: taking the first quartile leads to having at least a quarter of all the points set as core points with high probability, which will help covering a good portion of the dataset in few levels.

The border-detection procedure increases the r value at *epsStep* increments. This is the only parameter, whose main impact is on the execution time: very small steps lead to many iterations to converge. In our experiments, we found that a value of 10^{-3} was reasonable for the hardware at our disposal.

The pseudo-code for the *epsilon* self-tuning is reported in Figure 5.2. Since the data partitions are independent, the main loop (Figure 5.2, lines (1)-(1)) is executed in a distributed fashion by exploiting Spark (each data partition is associated with an independent task).

Algorithm 2 MinPoints setting

```
Input  : Dataset partitions - dataPartitions
Input  : Epsilon - epsilon
Output : Estimate of best MinPoints - bestMinPoints

Map<Int,Int> histogramNeighbours = {}
for partition in dataPartitions do
    for p in partition do
        /* p is a core point if MinPoints is lower than or equal to
           the number of its neighbours */
        numNeighbours = numNeighboursRadiusEps(p, epsilon, partition)
        /* Update the statistics about the number of points with
           numNeighbours neighbours */
        histogramNeighbours[numNeighbours] =
        histogramNeighbours[numNeighbours] + 1
    end
end
Map<Int,Int> numOfCorePoints = {}
neighboursAfterMinPoints = 0    /* Given MinPoints, the number of
    core points is the number of points with more than MinPoints
    neighbours */
for MinPoints in histogramNeighbours.keys().sort().reverse() do
    numOfCorePoints[MinPoints] = histogramNeighbours[MinPoints]
    + neighboursAfterMinPoints; neighboursAfterMinPoints =
    numOfCorePoints[MinPoints]
end
max=0    /* Select the MinPoints value that maximizes MinPoints ×
    number of core points */
for MinPoints in histogramNeighbours.keys() do
    numCorePoints = numOfCorePoints[MinPoints]  /* d is the number of
        attributes */
    if (MinPoints > d and numCorePoints × MinPoints > max) then
        max=numCorePoints × MinPoints  bestMinPoints=MinPoints
    end
end
return bestMinPoints
```

Fig. 5.3 Automatic setting of the *MinPoints* parameter value.

c) *Automatic parameter setting: $MinPoints$*:

Once $epsilon$ has been set, the value of $MinPoints$ is automatically set by selecting the value of $MinPoints$ for which Eq.(5.1) is maximum. The approach stems from the following observations.

$$MinPoints \times \text{numberOfCorePoints}(dataset, MinPoints, epsilon) \qquad (5.1)$$

$MinPoints$ represents the minimum size of the generated clusters. Since small clusters are not interesting, because they represent a negligible part of our data, while we are interested in the main groups and their characterization, we aim at setting high values of $MinPoints$: The higher the value of $MinPoints$, the higher the minimum cardinality of the generated clusters. However, the higher the value of $MinPoints$, the lower the number of core points will be. With lower values of $numberOfCorePoints$, the amount of clustered data potentially decreases, while the amount of noise points increases. Since we are interested in clustering as many data points as possible, discarding only the minimum amount of objects in lower-density areas, we should consider high values of $numberOfCorePoints$. To balance the two discussed trends, we set the $MinPoints$ trade-off to the value that maximizes $MinPoints \times \text{numberOfCorePoints}(dataset, MinPoints, epsilon)$. Figure 5.3 reports the pseudo-code of the algorithm that automatically sets $MinPoints$ given the value of $epsilon$. Also in this case, the procedure can be parallelized by assigning each data partition to a different Spark task.

5.4.2 Cluster and Data Characterization

Since clusters are anonymous groups of network traffic flows, but human-readable results are much more valuable to domain experts, the SeLINA methodology, as reported in block (2) of Figure 5.1, is designed to enrich clusters with (i) general attribute-based statistics, and (ii) domain-specific knowledge, for each cluster in the resulting set, as detailed in the following. The former does not require user intervention, whereas the latter can be guided by domain experts, by a-priori selecting specific attributes of interest.

- *Number of flows.* It provides insights into the data distribution, by identifying clusters covering most of the dataset and others identifying small “remote” groups of traffic flows. For instance, some network datasets present a predominant cluster with regular traffic and many smaller clusters identifying deviations. Other datasets may present similarly-sized clusters, (i.e., with the same number of flows) for different subnets or services.

- *Top characterizing attributes.* SeLINA provides the attributes with the highest Variance Reduction Ratio (VRR) for each cluster with respect to their variance over the whole normalized dataset. Given an estimator for the variance $\hat{\sigma}^2$, the Variance Reduction Ratio (VRR) for the j-th cluster and i-th feature x^i is defined as follows.
$$VRR_j(x^i) = \frac{\hat{\sigma}_D^2(x^i) - \hat{\sigma}_j^2(x^i)}{\hat{\sigma}_D^2(x^i)} \tag{5.2}$$
where $\hat{\sigma}_D^2$ is the variance over the whole dataset and $\hat{\sigma}_j^2$ is the variance over the j-th cluster. The rationale behind the variance reduction is to quantify the information gain, for a given attribute, obtained by isolating some of the flows in a cluster; it is inherited from decision trees [122], where the order of the attributes in the tree influences performance and results. Together with the variance itself, VRR is a strong indicator of the features that characterize a cluster the most and their relative importance.

- *Network domain statistics.* Network-oriented features of interest provided by SeLINA are the number of different source IP addresses, ports, and service types per cluster. Furthermore, the current implementation of SeLINA computes and plots the Cumulative Density Function (CDF) of selected dataset attributes (see Table 5.2 and Sec. 5.6 for details). For instance, per-cluster statistics of server IP addresses, server L4-ports, L7-application protocols, etc., are provided. Such attributes, despite being discarded during the clustering, are often crucial to allow domain experts to correctly extract meaning from the results.

5.4.3 Classification Model Training

All flows processed by the clustering algorithm (excluding the final iteration noise points) are labelled according to their cluster (e.g., cluster 1, 2, 3), and form a labeled dataset (i.e., a training set), which can be exploited for supervised learning. Thus, the goal of this phase, depicted in block (3) of Figure 5.1, is to build a classifier to efficiently label new unseen flows as they are captured.

Even if the cluster set could be directly exploited for labeling unseen data, a new ad-hoc classifier is trained separately to reach two design goals: (i) to provide a real-time high-performance classifier, and (ii) to build a human-readable model that can harness the knowledge inside the data.

To this aim, SeLINA exploits decision trees [123] to build the classification model. They are a well-known popular and mature technique able to reach both good accuracy and easy model interpretability, with the latter being a highly-valued feature for domain experts. To provide the intuition of how a decision tree works, we describe a toy example in the following.

Tree Example

Table 5.1 shows a simple training set with 5 records, each characterized by two attributes. Two clusters/classes are present (Cl. 1 and Cl. 2). A possible decision tree is reported in Figure 5.4. The node labels represent a feature (e.g., the size of the flow in bytes), while each branch is labelled with a possible value, or a range of values, for the feature within the node. In our example, the split from the root node is done on a range of values of the minimum round trip time. Each path from the root node to a leaf node represents a rule characterizing a class (a cluster in our case). The path within the dotted box models the simple rule

$RTT < 5ms \rightarrow cluster1$, thus this leaf can be interpreted by the analyst as a set of flows served by a cache in the nearby, with cluster 1 partly served by this cache, which serves uniquely this cluster. This kind of information is human-readable and provides a good characterization of how the traffic labeling is performed.

Knowledge Model

The output tree provides an easy-to-read overview of the features that best split the dataset according to the labels: for each node of the tree the split criterion can be written as an if/else condition over a single feature and a splitting value, and few

Table 5.1 A toy dataset

Id	$RTT[ms]$	$DataByte$	Class
1	3	2M	Cl. 1
2	20	900k	Cl. 2
3	12	1.5M	Cl. 1
4	15	500k	Cl. 2
5	12	3M	Cl. 1

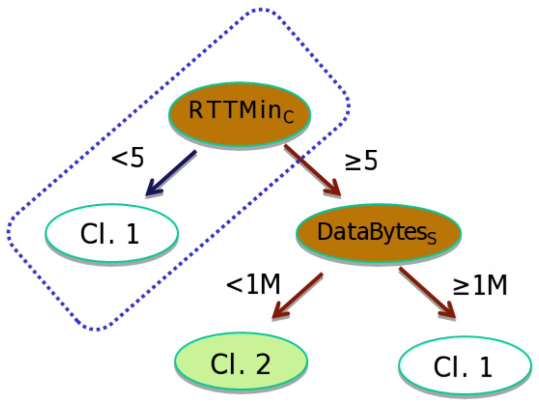


Fig. 5.4 A toy example of a decision tree.

levels of the tree are usually sufficient to show the most significant splits for the purpose of the classification.

Split Criterion

In the current work, the *Gini index* impurity-based criterion has been used to grow the tree. The Gini index is among the most popular choices and typically yields high-quality results. we exploited the Spark decision-tree implementation, which provides both the Gini and the entropy criteria. we performed some experiments, not reported here for the sake of space, to compare the accuracy of the classification models based on the Gini and the entropy indices and their results are very similar. we defer the reader to [124] for details about the Gini and the entropy indices.

5.5 Online characterization and model update

This component analyzes new network data in real-time by applying the model built in the previous block. As depicted in Figure 5.1, it consists of two phases: a real-time data labeling phase (4), and an online characterization phase (5).

The classification model is exploited in the real-time data labeling phase of step (4), where each new network flow is assigned a label.

Then, at step (5) the quality index computation is executed, to self-assess the model fitting and its results over time. When the quality index falls below a given threshold, the offline model building can be automatically triggered to rebuild a model better fitting the new data, thus providing self-evolutionary features. While the online data labeling phase (4) is straightforward, as it consists of a classification model application, in the following we provide details on the quality index computation in step (5) and the self-evolution policy stemming from such quality evaluation.

5.5.1 Quality Index

When no external information is provided (e.g., ground-truth class labels), the clustering results are evaluated on the shape of the clusters themselves. To this purpose, SeLINA exploits a well-known quality index, named *Silhouette* [125]. This index measures both intra-cluster cohesion and inter-cluster separation to evaluate the appropriateness of the assignment of a data object to a cluster rather than to another one.

Let $\mathbb{C} = \{C_1, \ldots, C_n\}$ be a set of clusters. The Silhouette value for a given data object r_i in a cluster $C_k \in \mathbb{C}$, given a distance measure d, is computed as

$$s(r_i) = \frac{b(r_i) - a(r_i)}{\max\{a(r_i), b(r_i)\}}, \tag{5.3}$$

where $a(r_i)$ is the average distance of object r_i from all other objects in cluster C_k, i.e.

$$a(r_i) = \frac{1}{|C_k|} \Sigma_{r_j \in C_k} d(r_j, r_i) \tag{5.4}$$

and $b(r_i)$ is the lowest average distance from all other clusters, i.e.

$$b(r_i) = \min_{C_l \in \mathbb{C}} \left(\frac{1}{|C_l|} \Sigma_{r_j \in C_l} d(r_j, r_i)\right), \forall C_l \neq C_k. \tag{5.5}$$

The Silhouette value for an arbitrary cluster $C_k \in \mathbb{C}$ is the average Silhouette value on all objects in C_k. It is computed as

$$s(C_k) = \frac{1}{|C_k|} \Sigma_{r_i \in C_k} s(r_i) \tag{5.6}$$

Lastly, the average $s(r_i)$ over all data of the entire dataset is a measure of how appropriately the data has been clustered. The distance measure d must be the same used for clustering, thus the Euclidean distance in our case.

The Silhouette coefficients take values in $[-1, 1]$. Negative and positive Silhouette values represent wrong and good object placements, respectively. Hence, the ideal clustering algorithm splits the data in a set of clusters $\mathbb{C}$ such that all clusters in $\mathbb{C}$ have a Silhouette value equal to 1. However, Silhouette values around 0.5 are already considered very high values representing a strong clustering result [125].

5.5.2 Characterization and Self-Evolution Policy

As the quality of the network traffic model is subject to ageing, SeLINA continuously evaluates the quality degradation of the model itself, with a two-fold objective: (i) highlighting substantial changes in the network and (ii) triggering the regeneration of the model as soon as the quality index falls below a threshold.

Since SeLINA computes the Silhouette for each new flow against the ones seen during the training phase, this quality index indicates how well the new flow fits the old clusters. A Silhouette close to 1 would indeed mean that the intra-cluster distances are negligible compared to the inter-cluster ones. The Silhouette values for the clusters ($s(C_j)$) are recomputed every N new records, where N is set by the user. The Silhouette index for the clusters significantly changes as soon as new kinds of data (not seen during the training) are added to the input (see Section 5.8.2 for an example). Unseen values should indeed get a Silhouette close to 0, while mispredictions, i.e., assignments to the wrong cluster, would have a negative value.

Besides the Silhouette indicator, SeLINA also tracks the number (percentage) of new flows assigned to each cluster over time. This helps in detecting changes in the traffic characterization due to (i) degradation in the clustering quality and (ii) shift in the distribution of the traffic flows among different clusters, as discussed in the experiments.

The final goals of the real time evaluation are to keep track of the state of the network, to identify changes and react. The reaction strongly depends on the use case and on the type of change. When SeLINA is trained on a standard behaviour, e.g. a usual working day without interruptions of service or congestion, a change is a strong hint of an anomaly, a strong congestion or an attack. The identification can be performed by looking at the current clusters and their cardinality in recent time frames. If the Silhouette value is unchanged, the current clusters do still model well the traffic, and the anomaly occurs only in the distribution of the flows among the clusters. If the Silhouette value of one or more clusters decreases, instead, the change is way more significant: the current model cannot describe the traffic anymore. The inspection in this case needs a new clustering, which can also be automatically triggered by the system. The new clustering can be executed on the whole historical dataset or on the most recent flows only. The latter option generates a more specific up-to-date model, that could be less general due to fewer training data.

5.6 Experiments and datasets

we experimentally evaluated SeLINA on two real network traffic datasets, associated with two different use cases. My goal is to show how SeLINA (i) effectively characterizes network traffic, and (ii) supports the analyst in understanding changes of the traffic mix. we focused on two real-world use cases. The first one consists of a dataset of YouTube flows in which we know the CDN had changed over time, causing possible issues to both end-users and ISPs [88]. The second case deals with the understanding of P2P traffic, for which, instead, little knowledge is available. In both cases, SeLINA autonomously extracts information from the automatic analysis of the traffic summaries, and presents results to domain experts in an interpretable format.

Metric	Description	Intuition
$L7-Data$	Amount of application payload transferred	Identifies possible different type of flows
$Duration$	Time since the first SYN to the last segment	Related to performance issues, and type of flow
$RTTMin$	Per-flow minimum RTT	Estimate of the “distance” between client and server
P_{reord}	Per-flow reordering probability	Identifies possible packet losses occurred before the probe
P_{dup}	Per-flow duplicate probability	Identifies possible packet losses occurred after the probe

Table 5.2 Features used by SeLINA as input.

We collected network traffic data through a passive probe running Tstat as explained in Sec. 1.1. The passive probe sniffs all packets flowing on the link and exploit Tstat to passive monitoring all TCP (and UDP) flows.

In this work, we focus on two datasets, collected during two different time periods. The first one consists of flows carrying YouTube videos. The second one collects all TCP flows excluding web traffic, i.e., it consists of mostly P2P traffic. I refer to each dataset as “YouTube” and “P2P” in the following.

The YouTube dataset consists of TCP flows collected during May 2013 by a probe placed on a PoP of a nation-wide ISP in Italy where the traffic aggregate from more than 10,000 customers is monitored. we use data from May 1st, 2013 to let SeLINA build the offline model. Datasets from May 2nd to May 31st are used instead to run the online model update phase and highlight traffic changes possibly suggesting the automatic model rebuilding. For this dataset, we know that during the second part of May 2013 the YouTube CDN had relevant changes affecting end-user quality of experience [88, 3]. Hence, we consider this as ground-truth information that allows us to verify if SeLINA correctly identifies interesting events.

The P2P dataset refers to April 17, 2012. From it we extract all TCP flows whose application protocol is neither HTTP nor HTTPS, i.e., where the majority of the traffic is due to P2P applications [126]. Traffic comes again from a backbone link of a nation-wide ISP in Italy.

Among the measurements exposed by Tstat, we consider the metrics reported in Table 5.2. we selected them since they are correlated to both system configuration and possible performance issues. For instance, the measure of the Round Trip Time (RTT) is related to both the distance from the server, and possible congestion on the path. Similarly, both reordering and duplicate probabilities increase during periods of congestion. Finally, duration and amount of carried data are possibly linked to the type of service the flow carries, e.g., short-lived signaling flows carrying little data rather than long lived data flows carrying a large amount of data. Since

SeLINA model building is based on unsupervised clustering, we expect the system to automatically leverage information offered by these features to identify proper classes of flows.

The metrics are computed by observing the TCP headers, and correlating them with information in the corresponding TCP ACKs. For instance, P_{reord} and P_{dup} are computed by keeping track of TCP sequence number evolution over time, while $RTTMin$ is the minimum delay observed between a data segment and the corresponding acknowledgement. Since TCP offers a bidirectional service, we consider measurements for each half-flow, i.e., segments from the client to the server, and vice versa. I denote them in the following by adding a subscript C or S for client or server side, respectively. For instance $RTTMin_S$ is the minimum delay observed at the probe between segments sent by the server and ACKs sent by the client, i.e., it is the delay between the probe and the customer client – the access network delay. Conversely, $RTTMin_C$ measures the time since the probe observes the client segment and the server ACK, i.e., it is the minimum RTT between the probe and the server – the backbone network delay.

Additional attributes and measures are included in the final results flows aggregated in the same cluster. Clusters are annotated by SeLINA before being presented to the domain experts. The additional features are not considered during the model building phase. For instance, once the cluster is built, the system computes Cumulative Density Functions (CDF), average, percentiles, etc. of the per-metric distribution of information extracted directly from features.

The datasets have been stored in a cluster at our University running Cloudera Distribution of Apache Hadoop (CDH5.3.1). All experiments have been performed on our cluster, which has 30 worker nodes, and runs Spark 1.2.0, HDFS 2.5.0, and Yarn 2.5.0. The cluster has a total of 2.5TB of RAM, 324 cores, and 773TB of secondary memory. The current implementation of SeLINA is a project developed in Scala exploiting the Apache Spark framework.

5.7 YouTube use case

In this section we discuss the network traffic characterization of the YouTube dataset first, as a result of the offline SeLINA component, and then we present an evaluation of the online part. The default values of $EpsStep$=0.001 and 3 clustering levels led to meaningful results for this experiment. Increasing the number of levels brings no improvement. After the third iteration, new clusters become very small and have very low Silhouette values, a clear sign that the system is artificially aggregating data that are actually very fragmented.

5.7.1 Offline Cluster and Model Characterization

Clustering results provide meaningful insights into network traffic when enriched by means of relevant statistics and features. As such, we present traffic analyses provided by both the cluster statistics and the classification model.

Cluster Statistics

Table 5.3 reports the clusters obtained by running SMDBSCAN on the YouTube dataset of May 1st, 2013. For each cluster, SeLINA returns the top-3 attributes according to VRR, i.e., it presents to the network analyst those features that best represent the data inside the cluster itself. For instance, consider the cluster number 1. It is the biggest one, collecting approximately 60,000 (36%) flows. It is primarily characterized by a rather low P_{dup} value (0.65%±0.71%), and clients requesting 4kB of data on average ($L7-Data_C$=3992±2422.4 bytes), a rather sizable HTTP request size. $RTTMin_S$ is 33.7±16.1 ms, which suggests quite standard and not congested DSL lines. The cluster thus collects the most common flows. This is the only cluster identified during the first iteration of the multi-level clustering. During iteration 2 and 3, more clusters emerge (one in step 2, and two in step 3), each with several thousands of flows. This confirms the ability of SMDBSCAN to identify large clusters, despite different densities, thanks to the multi-level approach. At the end of the whole clustering process, the noise cluster aggregates all remaining points. There are 40,000 of them (23%), which are very sparse, as proven by the high variance in their characterizing attributes.

Clusters 2 and 3 represent a sizable part of the traffic, with 16% and 22% of the flows, respectively. Interestingly, those are characterized by two very different $RTTMin_C$ values. Recall that $RTTMin_C$ represents the distance of the YouTube CDN server to the probe. Servers in Cluster 2 are 25.3±1.4 ms far, while servers in Cluster 3 are much closer (5.4±4 ms). P_{dup} is significantly different too, with

Table 5.3 YouTube dataset. Cluster characterization.

Lev.	Cl. id	Num. flows	Top-3 representative attributes ranked by highest variance reduction ratio		
			Attribute	Avg. value	Std. dev
1	1	59846	P_{dup}	0.65%	7.12E-03
			$L7-Data_C$	3992.1	2422.4
			$RTTMin_S$	33.7	16.1
2	2	27158	$RTTMin_C$	25.3	1.4
			P_{dup}	0.55%	6.42E-03
			$L7-Data_C$	5357.8	2916.9
3	3	37964	P_{dup}	2.97%	2.31E-02
			$RTTMin_C$	5.4	4.0
			$RTTMin_S$	52.6	42.2
	4	5569	$RTTMin_C$	25.5	1.2
			P_{dup}	4.11%	1.28E-02
			$Duration$	51464.0	31969.4
noise		40318	P_{reord}	0.000002%	3.12E-06
			$L7-Data_S$	14465449.3	30919388.4
			$RTTMin_S$	78.7	160.6

Cluster 2 P_{dup} being one order of magnitude smaller than cluster 3. This probably reflects higher congestion in the path from the probe to the client in cluster 2.

Cluster 4 collects fewer points (5,500, 4%). P_{dup} and $RTTMin_C$ are similar to Cluster 2, but here duration (51±32 s) is very large. This possibly hints for TCP flows of long lived video sessions. we will get back to this when observing the video resolution distribution in the following.

Besides the top-3 attributes selected for each cluster, SeLINA offers to the analyst a further characterization of the network traffic by presenting CDFs of features and additional measurements collected by the probe. In this use case, we consider the distribution of the RTT, of the throughput, and of the type of video format and resolution.[3]

Fig. 5.5 and Fig. 5.6 report the CDF of the $RTTMin_C$ and the average download throughput for each cluster. Cluster 2 and 4 show similar distributions of the $RTTMin_C$, as previously discussed, which are significantly different from the clusters

[3]Tstat has a DPI engine specialized in extracting metadata from YouTube flows.

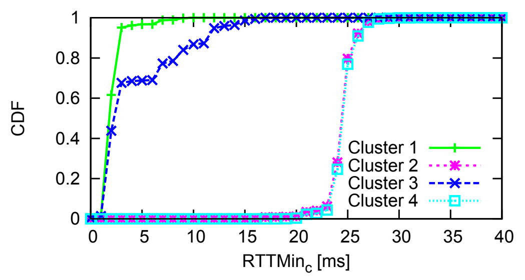


Fig. 5.5 YouTube dataset. $RTTMin_C$ distribution for each cluster.

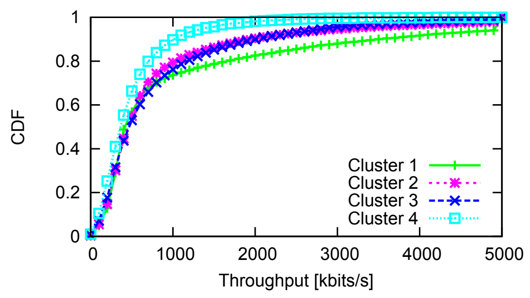


Fig. 5.6 YouTube dataset. Throughput distribution for each cluster.

1 and 3, whose flows, instead, are characterized by generally low values of $RTTMin_C$ (i.e., they represent requests served by nearby CDN servers). On the contrary, the flows of clusters 2 and 4 are associated with video requests that are served by relatively far CDN servers. Figure 5.6 shows that cluster 4 is also characterized by worse performance in terms of throughput, and it probably represents flows with possible performance issues.

This reflects the typical scenario of the YouTube CDN [88], and proves the ability of SeLINA to provide insights on the traffic mix. The analyst is offered few and consistent clusters, instead of thousands of single measurements.

To investigate further the characteristics of each cluster, Table 5.4 details the percentage of flows per cluster for each video resolution format. For each cluster, the 3 most frequent formats are reported. Each format is characterized by the quality of the video, mainly in terms of resolution (e.g., 240p, 720p), and it is identified by an integer value.[4] Some formats are shared by all clusters (i.e., format id 34 and 134), whereas others are peculiar for specific clusters, such as format 25 for cluster 1, and

[4]Video Resolution information at https://en.wikipedia.org/wiki/YouTube.

Table 5.4 YouTube dataset. Clusters' characterization based on video format

Lev.	Cl. id	Top-3 format ranked by number of flows	
		Format id (resolution)	Num. of flows (%)
1	1	25	63.71%
		34 (360p)	10.29%
		134 (360p [DASH])	7.36%
2	2	34 (360p)	60.67%
		134 (360p [DASH])	9.02%
		35 (480p)	8.35%
3	3	34 (360p)	55.12%
		35 (480p)	9.34%
		134 (360p [DASH])	9.22%
	4	34 (360p)	50.26%
		140 (AAC 128)	10.56%
		134 (360p [DASH])	9.43%

format 140 for cluster 4. It is interesting to notice that neither the format id, nor any other video information was exploited during the clustering phase, nevertheless the clusters are correlated to a set of video formats. For cluster 4, for instance, the longer *Duration* is due to higher presence of high quality audio stream server in format 140 (AAC 128kbps) that is not found in other clusters.

Classification Rules

The decision-tree described in Section 5.4.3 and trained with a maximal depth of 4 levels has been evaluated with a 3-fold cross-validation scheme on the training data. The average accuracy over the three cross-validation runs is 93%, and results for precision and recall for each class are shown in Table 5.5. All clusters are extremely well represented by the model for both precision and recall (93% to 99%), apart from a lower value in Cluster 1 recall (80%).

Being the model so accurate, rules that form the decision tree can be used to understand how the different clusters split the network traffic. Each path from the root to one leaf of the decision tree is translated into a rule for the class of that leaf. Each rule characterizes the data of its class (i.e., a cluster in our case). Rule-based

Table 5.5 Quality of the classification algorithm. 3-fold cross-validation

Cluster id	Precision	Recall
1	96.28%	80.17%
2	97.73%	93.02%
3	97.91%	99.25%
4	88.93%	98.22%

modeling provides further insights into correlations among attributes. Analying the rules of such classifier, we observe the following:

- $\{(RTTMin_C > 15.7ms)\ and\ (P_{dup} > 2.5\%)\} \rightarrow Cluster4$. This is the only rule associated with cluster 4. It states that all flows of cluster 4 are simultaneously characterized by a high $RTTMin_C$ and a high P_{dup}.
- $\{(RTTMin_C > 21.5ms)\ AND\ (P_{dup} \leq 2.5\%)\} \rightarrow Cluster2$. This rule provides a characterization of cluster 2, where flows have an even higher $RTTMin_C$ but a lower P_{dup} with respect to cluster 4.

Insights provided by such rules are relevant since we would not have been able to distinguish the differences between clusters 2 and 4 by considering only the CDF of $RTTMin_C$ reported in Figure 5.5. The rules, which simultaneously consider more than one measure, allow supporting domain experts to more easily characterize the content of the clusters and also perform comparisons among them by considering at the same time many facets.

5.7.2 Online Data Characterization and Model Update

The decision tree model is exploited to assign new flows, in real time, to the most appropriate class (which is one of the clusters). Every N assignments, SeLINA evaluates the quality of the current cluster set, by means of the Silhouette quality index and the distribution in number of flows assigned to each cluster. These two indicators provide the self-evolution feature to SeLINA, which is able to trigger a model rebuilding phase. The analysis of the Silhouette index indicates whether the classifier model does not fit the current data anymore, and the distribution of the flows among the clusters indicates whether the traffic patterns are changing. This information is also valuable for the analyst since it reflects changes in the traffic mix.

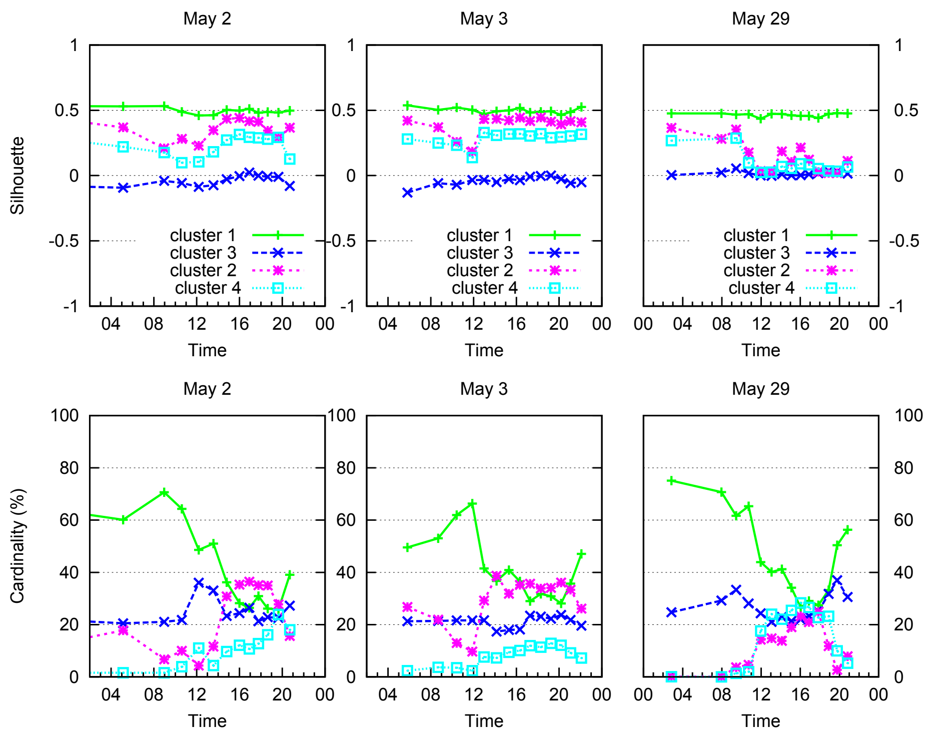


Fig. 5.7 YouTube dataset. Real-time data labeling: Silhouette and percentage of new flows assigned to each cluster.

The upper part of Figure 5.7 reports the value of the Silhouette index for three different days (May 2nd, May 3rd, and May 29th from left to right). The lower part reports the percentage of flows assigned to each cluster over time. The values are computed every $N = 10,000$ flows, which corresponds to 2-3 hours at night, and approximately 1 hour during peak traffic hours. Recall that the model had been trained on the May 1st dataset. Traffic from following days is assigned to clusters based on the classifier, but without re-running the clustering itself.

As discussed by domain experts in [88], network traffic in the first part of May is very similar to May 1st. On the contrary, in the second part of May, a change in the YouTube CDN occurred. As such, we would expect the Silhouette to reflect this situation, especially during peak time when the traffic is more significant. This is indeed the case. The Silhouette value is rather stable for all clusters during the first days of May, of whom we reported here May 2nd and May 3rd, meaning that there are no important changes in the traffic with respect to the May 1st model. It still fits the new data. Only cluster 2 and 4 show temporary and limited drops in

Silhouette values from 10am to 12am, but at that time, they only account for very low percentages of the traffic (less than 10% each).

The Silhouette values of clusters 2 and 4 during May 29th, when the significant change in the YouTube CDN already occurred, drop suddenly from 12pm to 8pm of May 29th. At that time, a sizable amount of traffic is assigned to these two clusters ($\approx$ 20% each), but the clusters 2 and 4 do not fit the data anymore, so that new flows present un-modeled characteristics, and they fall into the low-Silhouette clusters. Interestingly, cluster 1 still has a high Silhouette value (and counts for 30-40% of the traffic), reflecting that not all traffic is affected by the change.

A detailed analysis of May 29th highlights a significant increase of the $RTTMin_C$ values for cluster 2 and cluster 4 flows. While in May 1st and May 2nd, almost all flows have an $RTTMin_C$ lower than 25ms, in May 29th there are many flows with $RTTMin_C$ from 80ms to 100ms. The increase of the $RTTMin_C$ values is associated with changes in the YouTube's CDN previously identified in [88]. In the model built by SeLINA on May 1st data, there are no clusters representing this traffic pattern. Thus, the sudden drop of the Silhouette values automatically highlights the changes and can be used to raise an alarm.

To better highlight the difference in SeLINA results, we ran a set of experiments considering the first and last five days of May. we aim at identifying groups of similar traffic days, by means of the Silhouette pattern. Fig. 5.8 shows the correlation matrix of the per-day Silhouette indexes, i.e., for each pair of days, we measure how similar the Silhouette trends are. we use the Pearson correlation coefficient [124] among the Silhouette values during two days: when close to 1, the Pearson correlation depicts a strong positive linear correlation; when the coefficient is close to -1, it highlights a negative correlation, and when it is close to 0, negligible correlation is found. Left plot of Fig. 5.8 clearly highlights that Cluster 1 Silhouette is always very similar among those days, and stable over time. Cluster 1 consistently represents the part of traffic not affected by the CDN change, and the model always fits the new data. Right plot of Fig. 5.8, instead, clearly shows that Cluster 4 exhibits two patterns over time: during the beginning of May, it is consistent with the model. But during the last part of May (after the CDN change), its Silhouette daily pattern becomes very different from before. These results prove the strong link between the change in the Silhouette trends and the change in the traffic patterns, and the validity of using the drop of the Silhouette as a trigger for the generation of a new model of the network.

Focusing on the percentage of new flows assigned to each cluster (bottom plots in Fig. 5.7), we can detect changes related to the CDN allocation policy. For all days, the distribution of the flows changes from the late morning till evening. In particular, Cluster 1 traffic, which is characterized by low $RTTMin_c$ values, decreases from 60-70% (at night) to 30-40% in the 10am-9pm period. On the contrary, the number of flows assigned to Clusters 2 and 4 increases from less than 5% to 20-30% each. Clusters 2 and 4 are characterized by higher $RTTMin_c$ values, i.e., traffic is now being served by far-away servers. The difference between the two days is that in May 2nd and May 3rd the Silhouette values are high and stable, hence the changes in the distribution of the flows among the clusters are meaningful. On the contrary, on May 29th the drop in Silhouette values means that clusters cannot be trusted, and thus a model rebuilding is required.

we let then SeLINA rebuild the whole model (clustering and classifier) on the flows of May 29th from 10am to 9pm. This leads to a new characterization of the network traffic, not shown here due to lack of space. 10 clusters (instead of 4) are identified, with very different characterizing features, both in terms of number of flows and statistical distribution of values. The new model includes some clusters with very high $RTTMin_C$ values, which means that the presence of new CDN servers that were not properly represented by the previous clusters are now covered. For example, one of the new clusters, which contains about 12,000 flows, is characterized by an average $RTTMin_C$ value of 99ms$\pm$11ms, a significantly higher value than those of the former clusters (see Fig. 5.5). These results are consistent with those in [88], and confirm the ability of SeLINA to automatically identify changes in traffic pattern. Moreover, SeLINA extracts clusters which fit the new data and provide insightful analyses of the network traffic evolution.

5.8 P2P use case

In this section we show how SeLINA can help characterize the flows of the P2P dataset. we recall that this dataset contains all the TCP flows captured by Tstat, except the HTTP and HTTPS ones. Also in this case, we executed the offline self-learning phase by using the default values of $EpsStep$=0.001 and the first 3 levels of clustering. Differently from the YouTube use case, here we have no ground truth at our disposal, and thus SeLINA is used as a data exploration tool.

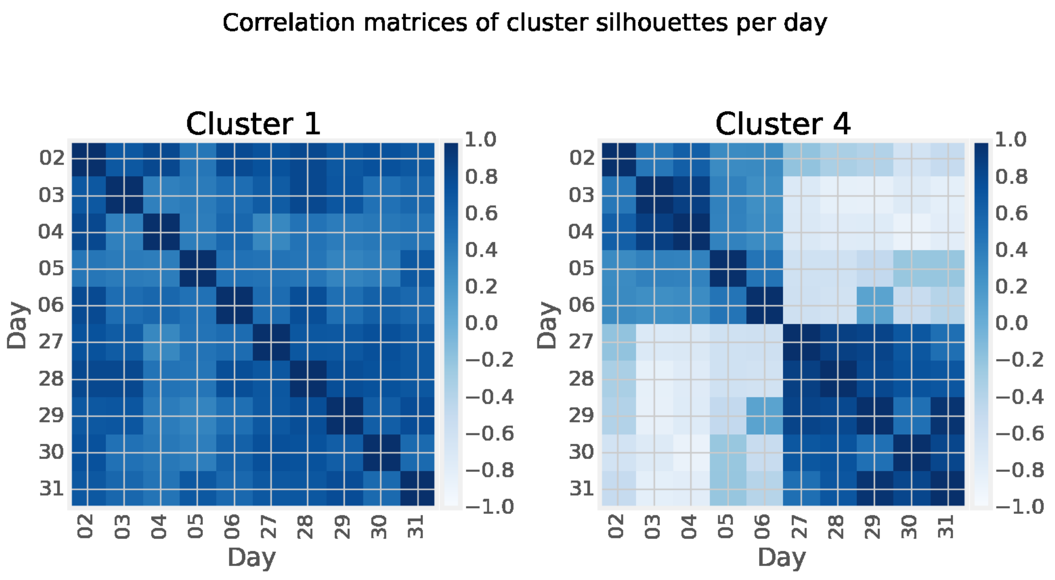


Fig. 5.8 YouTube dataset. Per-day correlation matrices of silhouettes for cluster 1 and 4.

5.8.1 Offline Cluster and Model Characterization

Table 5.6 reports the main characteristics of the extracted clusters and their top-3 characterizing attributes. we immediately notice a cluster with approximately 64% of the flows (Cluster 1, 98186 flows) that is significantly larger than any other cluster. Cluster 1 represents “standard” flows which are characterized by a similar average value of $RTTMIN_C$ and $RTTMIN_S$ (i.e. the communication time is similar in both directions of the flow). This balancing between $RTTMIN_C$ and $RTTMIN_S$ values is normal when no congestions are present. Indeed, P2P traffic is exchanged between residential clients, therefore, we can expect the distance between the probe and the peers to be somewhat equal, and so the RTT.

Other clusters provide interesting knowledge to domain experts as well. For instance, Cluster 2 has an average $RTTMin_S$ (235.5ms) that is two orders of magnitude higher than $RTTMin_C$ (9.4ms). These values highlight a significant asymmetry between the server side and the client side. The $RTTMin_S$ is very high (the average $RTTMin_S$ value of the ‘standard” flows in Cluster 1 is 33.5ms). This situation describes a possible congestion in one direction of the communication flow. we recall that we are analyzing P2P flows among ISP customers where many users are connected through an ADSL connection, with uplink capacity limited to 1Mbps.

Table 5.6 P2P dataset. Cluster characterization. Clusters obtained by using SMDB-SCAN and setting $EpsStep$=0.001

Lev.	Cl. id	Num. flows	Top-3 representative attributes Ranking by highest variance reduction ratio		
			Attribute	Avg. value	Std. dev
1	1	98186	$RTTMin_S$	33.5	38.6
			$RTTMin_C$	35.0	45.7
			$Duration$	33154.1	43104.8
2	2	15090	P_{dup}	3.30E-06	2.04E-04
			$RTTMin_S$	235.5	74.8
			$RTTMin_C$	9.4	22.2
	3	12152	P_{dup}	2.37E-05	5.44E-04
			$RTTMin_S$	14.6	22.9
			$RTTMin_C$	295.3	72.9
	4	4530	P_{dup}	4.44E-01	1.34E-03
			$RTTMin_S$	11.2	16.2
			$RTTMin_C$	18.5	12.9
3	5	3302	P_{dup}	1.31E-05	5.07E-04
			$RTTMin_S$	542.0	103.6
			$RTTMin_C$	5.2	13.6
	6	2524	P_{dup}	2.80E-01	2.49E-02
			$RTTMin_S$	6.9	15.0
			$RTTMin_C$	32.5	37.9
	7	1993	P_{dup}	1.05E-05	3.65E-04
			$RTTMin_S$	16.9	28.9
			$RTTMin_C$	608.1	101.5
	8	1892	P_{dup}	1.60E-01	1.88E-02
			$RTTMin_S$	14.7	29.6
			$RTTMin_C$	35.6	42.1
noise		13647	$Duration$	560334.5	4671692.5
			$L7-Data_S$	3585728.2	27938244.0
			P_{reord}	5.05E-03	2.90E-02

When a remote peer downloads a large amount of data from a local peer, the uplink of the latter may saturate, causing congestion (i.e., the average $RTTMin_S$ increases). On the contrary, the small $RTTMin_C$ suggests that the remote peer is connected via high speed FTTH technology (where access delay is much smaller being the upling capacity >10Mbps). A similar situation is valid also for the flows of Cluster

7. However, in Cluster 7 the $RTTMin_C$ is very high (608ms) and the $RTTMin_S$ is low (16.9ms). This second case reflects a simmetric scenario: high-speed local client downloading a lot of data from ADSL remote peers, whose uplink results congested.

5.8.2 Online Data Characterization and Model Update

For the P2P dataset we applied the evolving part of the framework to analyze how new flows are assigned to the clusters and identify possible changes in the type of traffic. Results present no significant changes in the Silhouette. This trend, that is confirmed by further statistics computed on the flows, highlights that there are no anomalies or changes in the traffic for the P2P dataset. The clusters identified by the clustering phase are still representative of the network flows, also of the future traffic. Since the day we analyzed is a "normal" one, SeLINA correctly identifies no changes and hence the re-execution of the clustering phase is not triggered.

5.9 Conclusion

This chapter presents a self-learning data analytics system that effectively mines network traffic data. The proposed methodology is based on a two-phase approach that

1. builds a self-evolving human-readable traffic model by autonomously splitting traffic data into homogeneous groups;
2. classifies new data in real-time and identifies the presence of changes in the traffic mix.

The SeLINA methodology features a distributed implementation in Apache Spark. It is a general purpose approach, which can be easily exploited to analyze network traffic data in different conditions. The approach has been tested in two real-world use cases. The performed experiments highlighted its ability to autonomously identify evolutions in the network and support the analyst by selecting characterizing features.

Possible extensions of the current work are (i) the inclusion of further cluster characterization measures, (ii) the evaluation of pre-processing feature selection techniques, and (iii) the design and integration of different analysis techniques, more appropriate for outlier detection.

Chapter 6

BGPStream: a software framework for live and historical BGP data analysis

6.1 Introduction

This chapter presents a paper titled *BGPStream: a software framework for live and historical BGP data analysis*. This paper has been published in the *ACM Internet Measurement Conference (IMC) 2016* conference.[1]

The work I present in this chapter has been developed with the team I joined during my period at CAIDA, a research center in the UCSD university. In particular I cooperate with them: (i) to perform big data analysis over BGP data (see Sec. 6.5), and (ii) to extend BGPStream features in order to store BGPStream data in a Kafka cluster (see Sec. 6.6.2).

We present BGPStream, an open-source software framework[2] for the analysis of historical and live Border Gateway Protocol (BGP) measurement data. Although BGP is a crucial operational component of the Internet infrastructure, and is the

[1] Chiara Orsini, Alistair King, Danilo Giordano, Vasileios Giotsas, and Alberto Dainotti, "BGP-Stream: a software framework for live and historical BGP data analysis" Proceedings of the Internet Measurement Conference (IMC). ACM, Santa Monica, California, USA — November 14 - 16, 2016, doi: 10.1145/2987443.2987482

[2] BGPStream is distributed with the GPL v2 license and is available at bgpstream.caida.org.

subject of fundamental research (in the areas of performance, security, topology, protocols, economy, etc.), there is no efficient and easy way of processing large amounts of BGP measurement data. BGPStream fills this gap by making available a set of APIs and tools for processing large amounts of live and historical data, thus supporting investigation of specific events, rapid prototyping, and building complex tools and efficient large-scale monitoring applications (e.g., detection of connectivity disruptions or BGP hijacking attacks). We discuss the goals and architecture of BGPStream and we show how the components of the framework can be used in different applicative scenarios.

6.2 Background

BGP Data at Router Level

The Border Gateway Protocol (BGP) is the de-facto standard inter-domain routing protocol for the Internet: its primary function is to exchange reachability information among Autonomous Systems (ASes) [127]. Each AS announces to the others, by means of BGP update messages, the routes to its local prefixes and the preferred routes learned from its neighbors. Such messages provide information about how a destination can be reached through an ordered list of AS hops, called an *AS path*.

A BGP router maintains this reachability information in the *Routing Information Base* (RIB) [127], which is structured in three sets:

- *Adj-RIBs-In*: routes learned from inbound update messages from its neighbors.

- *Loc-RIB*: routes selected from Adj-RIBs-In by applying local policies (e.g., shortest path, peering relationships with neighbors); the router will install these routes in its routing table to establish where to forward packets.
- *Adj-RIBs-Out*: routes selected from Loc-RIB, which the router will announce to its neighbors; for each neighbor the router creates a specific *Adj-RIB-Out* based on local policies (e.g., peering relationship).

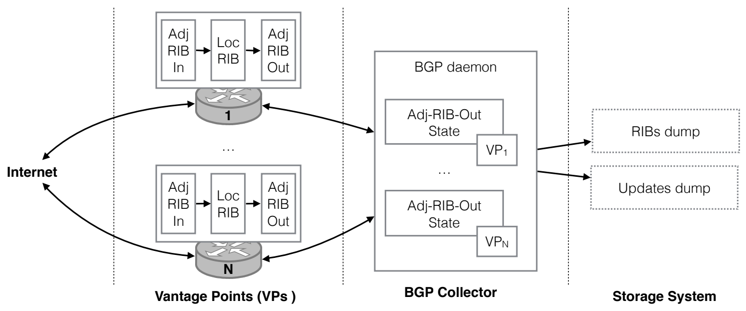


Fig. 6.1 BGP collection process.

BGP Data Collection

Some operators make BGP routing information from their routers available for monitoring, troubleshooting and research purposes. BGP *looking glasses* give users limited (e.g., read-only) access to a command line interface of a router, or allow them to download the ASCII output of the current state of the router RIB. Looking glasses are more useful for interactive exploration rather than systematic and continuous data acquisition. The latter can instead be implemented either *(i)* by establishing a BGP peering session with the monitored router from a dedicated system (a *route collector*), or *(ii)* through a protocol specifically designed for monitoring purposes, such as OpenBMP [128, 129]. OpenBMP is an open-source implementation of the BGP Monitoring Protocol defined in an IETF draft [129] and supported by latest versions of JunOS and Cisco IOS. The protocol allows a user to periodically access the Adj-RIBs-In of a router or to monitor its BGP peering sessions. While OpenBMP can be easily deployed within an AS to monitor its BGP routers, there are currently no projects which make such data publicly available. Route collectors instead, are often used for this purpose [130–132]. A route collector is a host running a collector process (e.g., Quagga [133]), which emulates a router and establishes BGP peering sessions with one or more real routers (*vantage points, VPs,* in the following).

Each VP sends to the collector update messages (*updates*) each time the Adj-RIB-out changes, reflecting changes to its Loc-RIB (Figure 6.1).

Once a BGP collector establishes a BGP session with a VP, it maintains a state and an image of the VP's Adj-RIB-out table derived from the updates received through the session. With different periodicity, it dumps (i) a snapshot of all the VP Adj-RIB-out tables (RIB dump) and (ii) the update messages received within that period from all the VPs (Updates dump).

Normally, a BGP session with a collector is configured as a *customer-provider* relationship, i.e., as if the VP was offering transit service to the collector. In this case, the VP is called *full-feed*, since it will advertise to the collector an Adj-RIB-Out which contains the entire set of routes in its Loc-RIB. This way, the collector potentially knows, at each instant, all the preferred-routes that the VP will use to reach the rest of the Internet – a partial view of the Internet topology graph visible to that router. A *partial-feed* VP instead, will provide through its Adj-RIB-Out only a subset of the routes in its Loc-RIB, e.g., routes to its own networks, or learned through its customers. Unfortunately, projects publicly providing information acquired by their collectors do not label VPs as full- or partial-feed, since peering with a collector is usually established on a voluntary basis and VP behavior can be subject to change without notice. Therefore, the policy that determines the Adj-RIB-Out to be shared with the collector must be dynamically inferred from the data (e.g., size of the Adj-RIB-Out).

For each VP, the collector maintains a session state and an image of the Adj-RIB-out table derived from updates. The collector periodically dumps, with a frequency of respectively few hours and few minutes, (i) a snapshot of the union of the maintained Adj-RIB-out tables (*RIB dump*) and (ii) the update messages received from all its VPs since the last dump, along with state changes (*Updates dump*). RIB dumps provide an efficient summary of changes to BGP routing tables with a coarse time granularity that is sufficient for several classes of studies [134–137]. In contrast, Updates dumps carry a lot of information to be processed, but offer a complete view of the observable routing dynamics, enabling other types of analysis and near-realtime monitoring applications [138–141].

Popular Data Sources

The most popular projects operating route collectors and making their dumps available in public archives are RouteViews [130] and RIPE RIS [131]. They currently operate 18 and 13 collectors respectively, which in total peer with approximately 380 and 600 VPs distributed worldwide (this number increases every year). Analyzing data from multiple VPs is of fundamental importance for most Internet studies, since each router has a limited view of the Internet topology and, even when full-feed, a VP shares only part of this information (the preferred routes). Moreover, macroscopic Internet phenomena visible through the routing infrastructure (e.g., outages, cyber attacks, peering relationships, performance issues, route leaks, router bugs) affect Internet routers differently, as a function of geography, topology, router operating system and hardware characteristics, operator, etc..

Such a distributed and detailed – even if partial – view of the inter-domain routing plane, generates large amounts of data (>2TB of compressed data collected in 2015 alone). RouteViews and RIPE RIS collectors save a RIB dump every 2 and 8 hours and an Updates dump every 15 and 5 minutes, respectively. Both projects save RIB and Updates dumps in a binary format, standardized by the IETF, called the Multi-Threaded Routing Toolkit (MRT) routing information export format [142]. RouteViews and RIPE RIS archives date back to 2001 and 1999 respectively, enabling longitudinal studies relevant to understand the evolution of the Internet infrastructure and its impact in other fields.

Software Frameworks and APIs

The most widely adopted software for BGP data analysis in the research community [143–149] is *libBGPdump* [150], an open source C library that provides a simple API to parse BGP dumps in MRT format and deserializes MRT records into custom data structures. It is distributed along with a command-line tool, *bgpdump*, that outputs MRT information read from a file in an ASCII format. Often researchers directly use the command-line tool to translate entire BGP dumps into text, and then parse the ASCII output to further process or archive the data. Although bgpdump has been an invaluable tool to support the analysis of BGP data over the last decade, it lacks the advanced features that we discuss in the next section (e.g., merging and sorting data from multiple files and data sources, supporting live processing, scalability, etc.).

There have been several projects that process BGP measurement data in real-time, developed both by industry (e.g., Dyn Research [151]) and academia (e.g., PHAS [152]), however their approaches are either undisclosed, or are specific to a certain application (i.e. they are not generalized frameworks). An exception is *BGPmon* [153, 154], a distributed monitoring system that retrieves BGP information by establishing BGP sessions with multiple ASes and that offers a live BGP data stream in the XML format (which also encapsulates the raw MRT data). Despite the fact that BGPmon enables rapid prototyping of live monitoring tools, it currently provides access to a limited number of VPs (compared to the vast number of VPs connected to RIS and RouteViews infrastructures), and it cannot be used for historical processing.

Towards Realtime Streaming of BGP Data

On the other hand, in the context of live monitoring, the major issue with popular public data sources such as RouteViews and RIPE RIS, is their file-based distribution system and thus the latency with which collected data is made available. Our measurements [155] show that, in addition to the 5 and 15 minutes delay due to file rotation duration, there is a small amount of variable delay due to publication infrastructure. However, 99% of Updates dumps in the last year were available in less than 20 minutes after the dump was begun. Since these latency values are low enough to enable several near-realtime monitoring applications, we began developing BGPStream with support for these data sources.

The research community recognizes the need for better support of live BGP measurement data collection and analysis. Since early 2015, we have been cooperating with other research groups and institutions (e.g., RouteViews, BGPMon, RIPE RIS) to coordinate efforts in this space [156]. Both RIPE RIS and BGPMon are developing a new BGP data streaming service (including investigating support for streamed MRT records), and BGPMon partners with RouteViews to include in the forthcoming next-generation BGPMon service all of their collectors. Experience with the development of BGPStream informed development efforts of the other research teams and vice-versa. While BGPStream is fully usable today, we envision that the forthcoming developments of these projects, likely deployed in 2016, will enhance BGPStream capabilities.

6.3 BGPStream Core

The BGPStream framework is organized in multiple layers (Figure 6.2). Blue boxes represent components of the framework; those marked with a star are distributed as open source in the current BGPStream release [157]. Orange boxes represent external projects or placeholders.

We discuss the core layers (*meta-data providers* and *libBGPStream*) in this section, whereas we illustrate the upper layers, through case studies, in the remainder

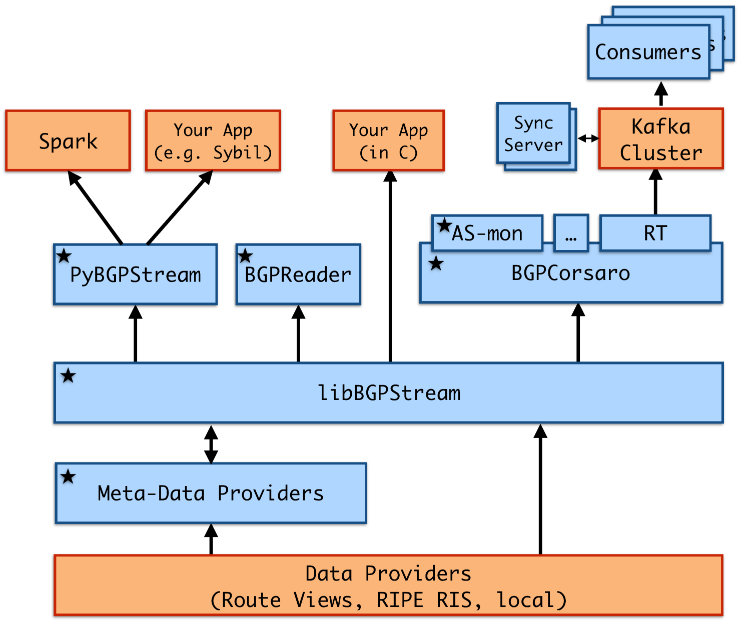


Fig. 6.2 BGPStream framework overview

of the chapter. Meta-data providers serve information about the availability and location of data from *data providers*, (either local or remote) which are data sources external to the BGPStream project.

libBGPStream, the main library of the framework (Section 6.3.3), provides the following functionalities: *(i)* transparent access to concurrent dumps from multiple collectors, of different collector projects, and of both RIB and Updates; *(ii)* live data processing; *(iii)* data extraction, annotation and error checking; *(iv)* generation of a time-ordered stream of BGP measurement data; *(iv)* an API through which the user can specify and receive a stream.

We distribute BGPStream with the following independent modules: *BGPReader*, a command-line tool that outputs the requested BGP data in ASCII format; *PyBGP-Stream*, Python bindings to the libBGPStream API; *BGPCorsaro*, a tool that uses a modular plugin architecture to extract statistics or aggregate data that are output at regular time bins.

6.3.1 Goals and Challenges

We designed the BGPStream framework with the following goals:

– *Efficiently deal with large amounts of distributed BGP data.* In Section 6.2, we emphasized the importance of performing analyses by taking advantage of a large number of globally distributed VPs. However, dealing with such large amounts of data as well as the distributed and diverse nature (different timing, formats, etc.) of the VPs pose a series of technical challenges.

– *Offer a time-ordered stream of data from heterogeneous sources.* BGPStream aims to provide a unified sorted stream of data from multiple collectors. Record-level sorting (rather than interleaving dump files) is important in at least two cases: (i) when analyzing long time intervals where time alignment cannot be achieved by buffering the entire input, and (ii) when an input data source provides a continuous stream of data (rather than a discrete dump file), since such a stream cannot be interleaved at the dump file level.

– *Support historical and near-realtime data processing.* We consider two modes of operation: *(i)* **historical** - all the BGP data requested is available before the program starts; *(ii)* **live** - the BGP data requested becomes available while the program is running. In live mode, the BGPStream stack plus the user application, must process data faster than it is generated by VPs/collectors. We minimize BGPStream processing latency, thus maximizing the time available for near-realtime user applications to perform live Internet monitoring and measurements (Sections 6.4.3 and 6.6).

Live mode also introduces the problem of sorting records from collectors that may publish data at variable times. This problem involves a trade-off between: (i) size of buffers, (ii) completeness of data available to the application, (iii) latency. Since such a trade-off should be evaluated depending on the specific goals and resources of the user application, we design BGPStream to perform best-effort record interleaving in live mode and we defer to the application the choice of a specific solution (in Section 6.6.2, we provide a concrete example of such a solution).

– *Target a broad range of applications and users.* Potential applications of BGPStream are both in the field of network monitoring and troubleshooting as well as scientific data analysis. The target user base should not be limited to the availability of high-performance computing and/or cluster infrastructure. The BGPStream framework makes available a set of tools and APIs that suit different applications and development paradigms (e.g., historical data analysis, rapid prototyping, scripting, live monitoring).

– *Scalability.* Since the pervasiveness of BGP VPs is key to monitoring and understanding the Internet infrastructure, the number of VPs supported by collector projects continually grows. In parallel, the technological challenges (e.g., near-realtime detection of sophisticated man-in-the-middle attacks [158, 159]) require solutions of increasing complexity and computational demand. We designed BGP-Stream to enable deployment in distributed and "big data analytics" environments: e.g., Spark's [22] native Python support makes BGPStream usable in such an environment out-of-the-box (Section 6.6.2).

– *Easily extensible.* Though our solution is designed to work with current standards and the most popular available data sources, we designed a stacked and modular framework, facilitating support for new technologies and data sources. BGPStream is indeed a project under evolution and is part of a coordinated effort with data providers, developers of complementary technologies, and users, to advance the state of the art in BGP monitoring and measurement data analysis [156, 160].

6.3.2 BGPStream Meta-Data Providers

One of the challenges in analyzing BGP measurement data is identifying and obtaining relevant data. Both RouteViews and RIPE RIS make data available over HTTP, with basic directory-listing style indexes into the data. Identifying the appropriate files for large-scale analysis (across multiple collectors and long time durations) involves either manual browsing and download, or scripting of a crawler tailored to the structure of each project's repository. Downloading the data, may itself take a significant amount of time (e.g., all data collected in 2014 is $\approx$2TB). Moreover, since both projects continually add new data to their archives as it is collected (Section 6.2), near-realtime monitoring requires custom scripts to periodically scrape the websites and download new data. BGPStream hides all of these complexities through meta-data providers: components that provide access to information about the files hosted by local or remote data repositories (the *Data Providers*, e.g., the RouteViews and RIPE RIS archives).

We implemented such a meta-data provider as a web service called *BGPStream Broker*, responsible for (i) providing meta-data to libBGPStream, (ii) load balancing, (iii) response windowing for overload protection, (iv) support for live data processing. The Broker continuously scrapes data provider repositories, stores meta-data about new files into an SQL database, and answers HTTP queries to identify the location of files matching a set of parameters. An instance of the Broker is hosted at the San Diego Supercomputer Center at UC San Diego and is queried by default by a libBGPStream installation, allowing BGPStream to be used "out-of-the-box" on any Internet-connected machine.

The Broker stores only meta-data about files available on the official repository, not the files themselves. This approach minimizes the potential for a bottleneck since queries to, and responses from, the Broker are lightweight, with the actual data being served by external data provider archives. This configuration also makes it simple to add support for additional data providers, as well as provide load-balancing and redundancy as the Broker can transparently round-robin amongst multiple mirror servers or adopt more sophisticated policies (e.g., requests sent from UC San Diego machines are normally pointed to campus mirrors).

While the Broker Data Interface is the primary data access interface, we also provide three other interfaces for analysis of local files: *Single file*, *CSV file*, and *SQLite*. The following sections assume that the Broker is used as the Data Interface.

6.3.3 libBGPStream

Application Programming Interface

The libBGPStream user API provides the essential functions to configure and consume a stream of BGP measurement data and a systematic organization of the BGP information into data structures. The API defines a BGP data stream by the following parameters: *collector projects (e.g., RouteViews, RIPE RIS), list of collectors, dump types* (RIB/Updates), *time interval start* and either *time interval end* or *live mode*. A stream can include dumps of different type and from different collector projects.

On the BGPStream website [161] we provide tutorials with sample code to use the BGPStream API. In general, any program using the libBGPStream C API consists of a stream configuration phase and a stream reading phase: first, the user defines the meta-data filters, then the iteratively requests new records to process from the stream. Code can be converted into a live monitoring process simply by setting the end of the time interval to *-1*.

Interface to Meta-Data and Data Providers

To access data and meta-data from the providers, the library implements a "client pull" model, which enables efficient data retrieval without potential input buffer overflow (i.e., data is only retrieved when the user is ready to process it).

To implement this model, the system iteratively alternates between making meta-data queries to the Broker and accessing and processing the dump files whose URLs are returned by the Broker. When the Broker returns an empty set of dump file URLs, the system signals to the user that the stream has ended. In live mode however, the query mechanism is blocking: if the Broker has no data available, libBGPStream will poll until a response from the Broker points to new data for processing.

Data structures and error checking

libBGPStream processes dump files [142] composed of *MRT records*. While an update message is stored in a single MRT record, RIB dumps require multiple records. The *BGPStream record* structure contains a de-serialized MRT record, as well as an error flag, and additional annotations related to the originating dump (e.g., project and collector names).

To open MRT dumps, we use a version of libBGPdump [150] that we extended to: (i) read remote paths (HTTP and HTTPS), (ii) support reading from multiple files in parallel from a single process, and (iii) signal a corrupted read. libBGPStream uses this signal to mark a record as *not-valid* (*status* field) when the BGP dump file cannot be opened or if the dump is corrupted. libBGPStream also marks records that *begin* or *end* a dump file, allowing users to collate records contained in a single RIB dump.

An MRT record (and therefore a BGPStream record) may group elements of the same type but related to different VPs or prefixes, such as routes to the same prefix from different VPs (in a RIB dump record), or announcements from the same VP, to multiple prefixes, but sharing a common path (in a Updates dump record). To provide access to individual elements, libBGPStream decomposes a record into a set of *BGPStream elem* structures. Table 6.1 shows the fields that comprise a BGPStream elem. The *AS path* field contains all information present in the underlying BGP message, as specified in RFC 4271 [127], including *AS_SET* and

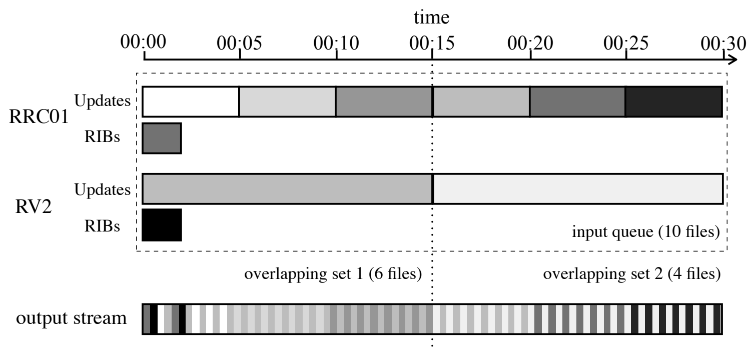


Fig. 6.3 Intra- and inter-collector sorting in libBGPStream.

AS_SEQUENCE segments. libBGPStream also provides convenience functions for easily iterating over segments in an AS path, accessing fields within a segment, and converting paths and segments to strings (using the same format as bgpdump). We do not currently expose all the BGP attributes contained in a MRT record in the BGPStream elem; we will implement the remaining attributes in a future release. The *old state* and *new state* fields refer to elems from RIPE RIS VPs. Each RIPE RIS collector maintains, for each VP, a Finite State Machine (FSM) for the status of the BGP session with the VP, I store the previous and current state of the FSM.

Field	Type	Function
type	enum	route from a RIB dump, announcement, withdrawal, or state message
time	long	timestamp of MRT record
peer address	struct	IP address of the VP
peer ASN	long	AS number of the VP
prefix*	struct	IP prefix
next hop*	struct	IP address of the next hop
AS path*	struct	AS path
community*	struct	community attribute
old state*	enum	FSM state (before the change)
new state*	enum	FSM state (after the change)

* denotes a field conditionally populated based on type

Table 6.1 BGPStream elem fields.

Generating a sorted stream

libBGPStream generates a stream of records sorted by the timestamps of the MRT records they encapsulate. Collectors write records in dump files with monotonically increasing timestamps. However, additional sorting is necessary when the stream is configured to include MRT records stored in files with overlapping time intervals[3], which occurs in two cases: *(i)* when reading dumps from more than one collector (inter-collector sorting); *(ii)* when a stream is configured to include both RIB and Updates dumps (intra-collector sorting). Since each file can be seen as an ordered queue of records, in practice libBGPStream performs a *multi-way merge* [162] on such queues.

Given the current number of collectors in RouteViews and RIS (about 30), and that the broker returns in each response a set of dump file URLs (*dump file set*) spanning up to 2 hours of data, the number of files to read can be up to $\approx$500. The computational cost of the multi-way merging is proportional to the number of queues (files) considered. We therefore break the dump file set in disjoint subsets, ensuring that we place files with overlapping time intervals in the same subset, and apply multi-way merge to each. This problem is exacerbated by the fact that the duration of Updates dumps vary between projects (e.g., RouteViews 15 min, RIPE RIS 5 min). We minimize the number of files per subset by iteratively applying the following steps until we process all files: *(1)* initialize a new subset with the oldest file in the set; *(2)* recursively add files with time intervals overlapping with at least one file already in the subset; *(3)* remove from the set the files in the subset. The subsets we obtain this way typically contain up to $\approx$150 files. For each of them, we perform the multi-way merge: libBGPStream simultaneously opens all the files in the set and iteratively *(i)* extracts the oldest MRT record from such files, and *(ii)* uses the MRT record to populate a BGPStream record.

Figure 6.3 shows an example of how RIB and Updates dumps generated by a RIPE RIS collector (RRC01) and a RouteViews collector (RV2) are interleaved into a sorted stream. The 30 minutes (10 files) of BGP data are first separated into two disjoint sets (of 6 and 4 files) based on overlapping file time intervals. Then a multi-way merge is applied separately to the two sets, yielding the stream depicted at the bottom.

[3]We define the time interval associated with a dump file as the time range covered by the timestamps of its records.

we empirically tested the cost of our sorting algorithm by using libBGPStream to process one day of Update and RIB dumps from all collectors of RouteViews and RIPE RIS, and confirmed that the cost of sorting is negligible compared to the cost of actually reading records from the dump files.

6.4 Analysis of specific events and phenomena

While users can write code that directly uses the BGPStream C API, we provide solutions that allow complex measurement experiments to be expressed with little to no code. We show how BGPReader and PyBGStream can rapidly address a variety of research tasks.

6.4.1 ASCII command-line tool

BGPReader is a tool to output in ASCII format the BGPStream records and elems matching a set of filters given via command-line options. This tool is meant to support exploratory or ad-hoc analysis using command line and scripting tools for parsing ASCII data. BGPReader can be thought of as a drop-in replacement of the analogous bgpdump tool (a command line option sets bgpdump output format), which is widely used by researchers and practitioners. However, BGPReader adds features such as the support to read data from multiple files, collectors, and projects in a single process, the ability to work in live mode and various filters.

For example, the following command line will dump on *stdout* a (sorted) stream of lines, each representing BGP updates from all the RouteViews and RIS collectors which are related to subprefixes of *192/8* and observed since May 12th 2016: `bgpreader -w 1463011200 -t updates -k 192.0.0.0/8`. The command will run indefinitely, outputting new data as it is made available by the data sources.

6.4.2 Python bindings

PyBGPStream is a Python package that exports all the functions and data structures provided by the libBGPStream C API. We bind directly to the C API instead of implementing the BGPStream functions in Python, in order to leverage both the flexibility of the Python language (and the large set of libraries and packages available) as well as the performance of the underlying C library. Even if an application implemented in Python using PyBGPStream would not achieve the same performance as an equivalent C implementation, PyBGPStream is an effective solution for: rapid prototyping, implementing programs that are not computationally demanding, or programs that are meant to be run offline (i.e., there are no time constraints associated with a live stream of data).

```python
from _pybgpstream import BGPStream, BGPRecord, BGPElem
from collections import defaultdict
from itertools import groupby
import networkx as nx

# create and configure new BGPStream instance
stream = BGPStream()
rec = BGPRecord()
# request RIB data from Aug 1 2015 7:50am -> 8:10am (UTC)
stream.add_filter('record-type', 'ribs')
stream.add_interval_filter(1438415400, 1438416600)
stream.start()

# create datastructures for the undirected graph and path
    lengths
as_graph = nx.Graph()
bgp_lens = defaultdict(lambda: defaultdict(lambda: None))

# consume records from the stream
while(stream.get_next_record(rec)):
    # process all elements of each record
    elem = rec.get_next_elem()
    while(elem):
        monitor = str(elem.peer_asn)
        # split the AS path into segments
        hops = [k for k,g in groupby(elem.fields['as-path'].
            split(" "))]
        # sanitization: ignore local routes
        if len(hops) > 1 and hops[0] == monitor:
            origin = hops[-1]
            # add all edges to the NetworkX graph
            for i in range(0,len(hops)-1):
                as_graph.add_edge(hops[i],hops[i+1])
            # how long this path is (for comparison to
                shortest path)
            bgp_lens[monitor][origin] = \
                min(filter(bool,[bgp_lens[monitor][origin], len(hops)]))
        elem = rec.get_next_elem()
# compare actual BGP path lengths to computed shortest path
for monitor in bgp_lens:
    for origin in bgp_lens[monitor]:
        nxlen = len(nx.shortest_path(as_graph, monitor, origin))
        print monitor, origin, bgp_lens[monitor][origin], nxlen
```

Listing 6.1 Calculate AS path inflation in ≈30 lines of code.

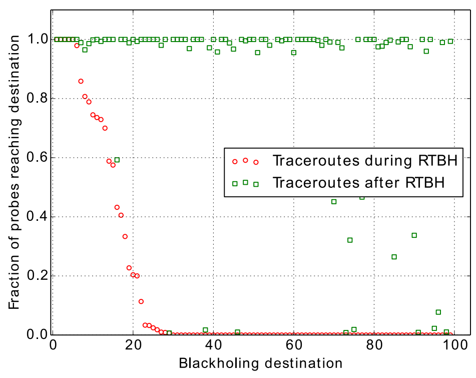
In Listing 6.1, we show a practical example related to a research topic commonly studied in literature: the AS path inflation [163]. The problem consists in quantifying the extent to which routing policies inflate the AS paths (i.e., how many AS paths are longer than the shortest path between two ASes due to the adoption of routing policies), and it has practical implications, as the phenomenon directly correlates to the increase in BGP convergence time [164]. In less than 30 lines of code, the program compares the AS-path length observed in a set of BGP RIB dumps and the corresponding shortest path computed on a simple undirected graph built using the AS adjacencies observed in the AS paths. The program reads the 8am RIB dumps provided by all RIS and RouteViews collectors on August 1st 2015, and extracts the minimum AS-path length observed between a VP and each origin AS. While reading the RIB dumps, the program also maintains the AS adjacencies observed in the AS path. We then use the NetworkX package [165] to build a simple undirected graph (i.e., a graph with no loops, where links are not directed) and we compute the shortest path between the same <VP, origin> AS pairs observed in the RIB dumps. In this example, we compare path lengths of 10M unique <VP, origin> AS pairs and find that, in more than 30% of cases, inflation of the path between the VP's AS and the origin AS accounts for 1 to 11 hops. Compared to the study of Gao and Wang, who analyzed RouteViews data from year 2000 and 2001 [163], these numbers show more inflated paths (>30% instead of >20%) and a consistent number of max additional hops (11 instead of 10).

In conclusion, this case study shows that BGPStream simplifies the task of analyzing heterogeneous BGP data, especially if we want such analysis to be systematically applied to different datasets and repeatable. To perform exactly the same experiment described here without using BGPStream, a researcher would have to manually identify and download all the data and write a parser for bgpdump ASCII output (besides writing the same core analysis logic). In addition, if a researcher wants to perform the same type of analysis on different sets of data (e.g., time window, subset of collectors, data collection projects) they would need to repeat the manual work of identifying and downloading all the files needed or, more realistically, develop a configurable crawler supporting the file hierarchies and naming conventions of different collection projects. Moreover, if another research task requires code to be aware of data collection time, type (RIB and Updates dumps) and provenance (collector, project), a researcher would need to build a data indexing system accessible to analysis code. While such efforts are doable, first they represent an added cost that greatly outweighs the cost of writing the core analysis code and deviates focus from the specific research task. Second, in a research context, similar efforts typically result in a mix of ad-hoc code and scripts that make sharing for reproducibility purposes improbable. Finally, since a script using PyBGPStream embeds the full definition of the input data used for an experiment, it further fosters reproducibility of experimental results.

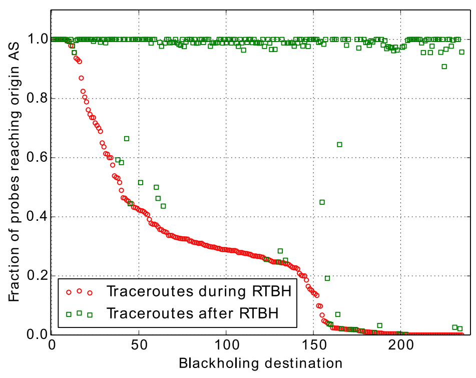
6.4.3 Timely Additional Measurements

In this section, we show how BGPStream can be combined with RIPE Atlas [166], a large-scale distributed infrastructure for active measurements, to enable timely analysis of customer-triggered BGP policies.

To mitigate the collateral damage of a DoS attack, a customer may use the Remotely Triggered Black-Holing (RTBH) technique to request its transit providers or peers to divert all the traffic towards its targeted IP addresses to a *null interface*, which drops all the incoming traffic [167]. Providers who support RTBH define a BGP community [168][169] that can be used by their customers to signal IP ranges to be black-holed. Since BGP communities lack standardization, the black-holing communities can differ among different providers [170], therefore multi-homed customers may need to set multiple black-holing communities to request black-holing from more than one of their providers. RTBH is effective on minimizing the collateral damage of DoS attacks, at the expense of taking the target completely offline. To limit the number of hosts adversely affected by black-holing, providers often restrict black-



(a) Fraction of traceroute queries that reach each black-holed destination.



(b) Fraction of traceroute queries per black-holed destination that reach each origin AS.

Fig. 6.4 Difference in the data-plane reachability of black-holed destinations during and after RTBH.

holing only to /32 prefixes, although shorter prefixes may also be allowed depending on each provider's policy. The lack of standardization regarding RTBH policies means that the impact of RTBH can be difficult to predict. Dietzel et. al. have studied the use of RTBH from the perspective of a large European IXP [171]. In this case study, we combine data-plane and control-plane measurements to demonstrate how we can gain a better understanding of how black-holing is implemented and its effects. Our purpose is to illustrate how BGPStream filters and live-mode streams facilitate complicated measurements that otherwise would require enormous instrumentation efforts, rather than providing a complete study of RTBH.

We identify as an RTBH request any triple of (collector, VP, prefix) that is tagged with at least one black-holing community from a list I compiled by parsing the IRR records and technical support websites for 30 ASes (13 Tier-1 providers, 12 multinational ISPs, and 5 academic networks). We respectively mark the *start* of an RTBH request when we first observe a BGP update with a black-holing community attached on a prefix that was previously announced without such a community, and the *end* when such prefix is re-advertised without it or explicitly withdrawn.

We executed our RTBH measurements between 20-29 April 2016 by continuously listening to BGP updates from the *route-views2* and *RRC12* collectors, for IPv4 prefix announcements tagged with black-holing communities. Almost 80% of the RTBH requests we detected have a duration of less than a day, while 20% have a duration of less than 40 minutes. These observations are consistent with previous studies on DoS attack duration [172, 171]. Therefore, it is important to minimize the delay between the application of black-holing communities and the detection time, in order to avoid missing the time window during which we can execute traceroute measurements toward the black-holed prefixes. To minimize latency between BGP and traceroute measurements, we utilize two BGPStream streams (within the same Python script) running in live mode to collect BGP updates. We apply community-based filters to the first stream so that it only yields prefix announcements tagged with at least one black-holing community. Whenever we observe a RTBH request from this stream, we add a filter for the black-holed prefix to the second stream to capture explicit or implicit withdrawals. Using two streams in this manner provides a clear separation of concerns, simplifying the logic in our Python script. That is, one stream triggers investigation of a prefix, whereas the other (possibly) triggers the completion of investigation.

Upon detecting the start of an RTBH request we orchestrate a set of *paris* ICMP traceroutes towards a random IP address in the corresponding prefix. Depending on the connectivity of the origin AS, we select between 50 to 100 currently-active RIPE Atlas probes from: (i) the visible AS neighbors of the origin AS, (ii) ASes that are co-located in the same IXPs as the origin AS, (iii) the same country of the target IP (to account for potentially invisible peripheral peering inter-connections). Our measurements are timely in most of the cases: we are able to probe over 95% and 90% of the black-holed prefixes, respectively for updates collected from RIPE RIS and RouteViews, before the RTBH is switched off. We also repeat the same traceroutes as we detect the end of the RTBH request after the blackholing community is withdrawn.

In total, we discovered 482 black-holed prefixes, originated by 67 different ASes. 398 of the black-holed prefixes had a length longer than /24, 397 of which had a length of /32 (single hosts). Contrary to the best practices that recommend the suppression of black-holed prefix advertisements [167, 173] or prefixes that are too specific [174], during the short period of our experiment we observed a non-trivial number of black-holed prefixes that propagated beyond the AS that defined the balck-holing communities. Namely, the corresponding ASes applied neither the egress filter for black-holed prefixes, nor the egress filter for too specific prefixes. Past works found that prefixes longer than /24 are visible to 20% – 30% of the monitors at the BGP collectors [175, 176]. In Section 6.5 we briefly analyze the propagation of BGP communities as it is visible from BGP collectors. However, the control-plane propagation of the black-holed prefixes beyond the network that applies the black-holing has not been analyzed before. From our measurement results, we remove prefixes for which I could not obtain traceroutes from the same set of Atlas probes between the two measurements (due to fluctuations in the availability of the probes), obtaining 253 prefixes that we briefly investigate. Figure 6.4 reports the difference in the data-plane reachability of black-holed destinations during (red) and after RTBH (green). For this representation, results are ordered based on tghe value of each metric during the RTBH.

Figure 6.4(a) shows the fraction of traceroutes that reach each destination. In this graph we do not include destinations that traceroutes do not reach after the RTBH, resulting in 100 destinations examined: after the RTBH 83% destinations are reached by at least 95% of the traceroutes, whereas during the RTBH 77% of the destinations are reached by less than 5% of the traceroutes and 73% are never reached by any probe. These numbers clearly show a change in data-plane reachability. Interestingly, during RTBH, 13% of the destinations are only partially reachable (between 20% and 80% reachability). For these destinations, we manually verified that traceroutes from customers or peers of the origin AS could still reach the black-holed destination, while ASes in the upstream path failed.

Although these results indicate a change in data-plane reachability during the RTBH events, the DoS attack itself may be the cause of traceroutes not reaching a host normally responding. However, if RTBH is in place the traffic is supposed to be dropped at the border routers [167, 177]. To dig deeper, in Figure 6.4(b) (where we consider all the 253 prefixes) we look at reachability at the level of the origin AS instead of the end host. On one hand, we find that the majority of the destinations (190) experience traceroutes that frequently fail (i.e., 40% reachability or less) to reach the origin AS. On the other hand, the vast majority of destinations show full reachability of the originAS after the RTBH. This preliminary finding is consistent with the expected behavior when RTBH is employed, and we plan to investigate it further in future work.

In conclusion, this case study shows that I are able to easily instrument combined passive control-plane and active data-plane measurements to capture and investigate transient routing policies. Without the capability to stream and filter BGP updates in near-realtime, we would be unable to capture the data-plane paths of the short-lived black-holing events. The alternative to BGPStream (or a system that implements the same features) would have been to continuously run traceroute scans against the entire address space which is impractical given the immense resource requirements of such an exhaustive probing.

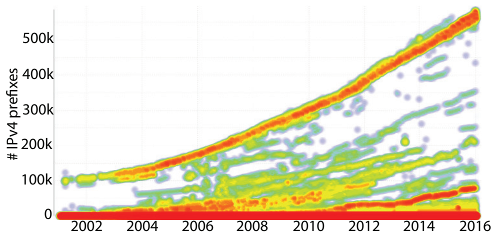
6.5 Analysis of Massive Datasets

In this section, we showcase simple case studies to demonstrate that BGPStream's Python bindings are readily usable in a big data workflow. we deploy PyBGPStream scripts in an Apache Spark [22] environment running on a 15-node cluster (240 CPUs and 960GB of RAM) to extract statistics of BGP across the last 15 years. we make these scripts available at [178] as a starting point for other researchers to use PyBGPStream with Spark for their own analyses. In addition, our examination highlights how certain features of the BGP eco-system (e.g., average size of the routing table) appear different depending on the data sources and data types picked from the heterogeneous BGP measurement infrastructure currently available to researchers, thus providing a reference for future research.

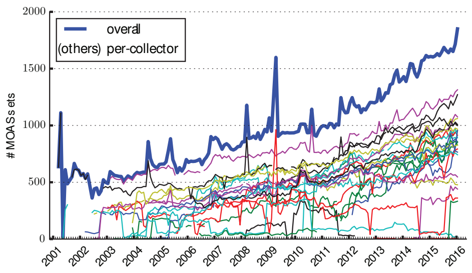
In all our analyses, I processed the midnight RIB dumps of the 15th day of each month from January 2001 to January 2016: more than 3000 RIB dumps, totalling approximately 44 billion BGP elems. The running times of the various analyses range between ≈1 and 24 hours. All our Python scripts share a common structure: *(i)* we build a list of data partitions splitting the data by time range and BGP collector and instruct Spark to create a Resilient Distributed Dataset (a data structure split across many nodes)[4]; *(ii)* we map a Python function to execute for every element in the list; this function represents the core of the BGPStream routines for data extraction; e.g., it creates the stream (defining filters etc.) and it executes nested *while* loops – per BGP record and per BGP elem – such as the one in Listing 6.1; this operation results in the creation of as many streams as list elements; *(iii)* we specify three independent reduction operations: per VP, per collector, overall. We also provide a *hello-world* template script that follows this pattern [178].

In our first analysis, we study the growth of the IPv4 routing table in BGP speakers over time (calculated as the number of unique prefixes in the Adj-RIB-out of each VP). There are three observations in this analysis useful as future reference for similar studies: *(i)* partial-feed VPs, i.e., those showing significantly smaller

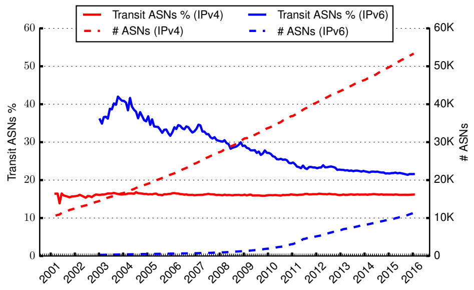
[4]We also specify the number of slices, typically 2-3 times the number of cores in the cluster.



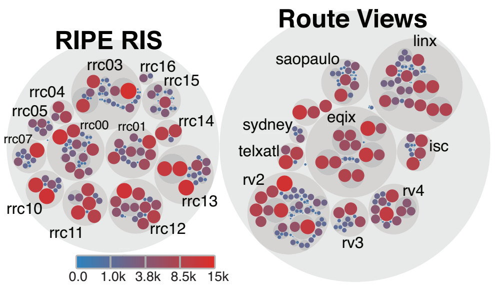
(a) heatmap depicting the growth of the IPv4 routing table in VPs over time.



(b) number of unique MOAS sets.



(c) absolute number and percentage of both IPv4 and IPv6 transit ASs.



(d) community iversity as observed by VPs

Fig. 6.5 Results of historical analysis using PyBGPStream and Apache Spark.

Adj-RIB-outs, are numerous and significantly skew the distribution; Figure 6.5(a) shows a heatmap of data from 2,296 VPs, the *y* axis shows the number of prefixes in the Adj-RIB-out of VPs; while warmer colors represent a higher concentration of points from different VPs. From this result we can see that only 710 out of 2,296 VPs are within 20 percentage points of the maximum at each time bin (we adopt this definition of full-feed VP in the following); *(ii)* two collectors (RouteViews *kixp* and *soxrs*) do not have a single full-feed peer, thus may not provide enough information for most analyses; *(iii)* we find that both the RouteViews and RIPE RIS repositories occasionally miss RIB dumps (34 per year on average) on midnight of the 1st day of the month (thus we perform our analyses with data from the 15th day of the month). In this analysis, we also compute, at each level of aggregation (VP, collector, overall), the number of unique prefixes and ASes observed, which we use to normalize data in the other analyses.

Figure 6.5(b) shows the results of analysis in which identified MOAS (Multi Origin AS) prefixes [179].

Study and detection of MOAS prefixes is relevant to many problems [140], including the detection of BGP hijacking activity [159]. Figure 6.5(b) plots on the y axis the number of unique sets of ASes (*MOAS sets* in the following) over time, contributing to MOAS prefixes aggregated into overall (top blue line) and per-collector (other lines). Besides the slow growth in observable MOAS sets over time, this graph highlights that to obtain a better view of MOAS prefixes, it is important to analyze data from as many collectors as are available: the number of MOAS sets identified in the *overall* aggregation is always significantly larger than the maximum number identified by a single collector.

We then calculated the number of transit ASes (ASes appearing in the middle of an AS path) observed for both IPv4 (red lines) and IPv6 (blue lines). Figure 6.5(c) shows that for IPv4, despite the nearly-linear growth in the number of ASes, the fraction of transit ASes over time has been constant! For IPv6 in contrast, overall there has been a constant decay in the fraction of transit ASes (edge growing faster than transit). However, around 2012, this decay slowed considerably, while the total number of IPv6 ASes kept a fast rate: the IPv6 graph is growing fast while its edge and transit portions recently started growing at similar paces! (Approaching the property we observed in the IPv4 graph over the last 15 years.) As of January 2016, however, the fraction of transit ASes is much larger in IPv6 (21% vs 16%), reflecting a smaller adoption of IPv6 at the edge.

In the final analysis we conducted with Spark, we investigated how BGP communities propagate and are visible via the RouteViews and RIPE RIS measurement infrastructures. BGP communities can be used to study several relevant Internet phenomena, such as complex AS relationships [135], traffic engineering policies [180], DDos mitication (Section 6.4.3). We collected unique communities appearing in IPv4 paths and we found that the number of observable communities over time increased from ≈800 (January 2001) to ≈40,000 (January 2016). In the rest of this section I focus on the most recent data (January 2016). By counting only the AS identifier portion of each community (which typically refers to the AS targeted by or generating the community), we observed approximately 4,000 ASes using communities. We observe communities only through ≈83% of the VPs, showing that many BGP speakers strip out communities from AS paths before propagating them. By observing the full paths, we find that at least 1,000 ASes propagate BGP communities (out of the more than 8,000 transit ASes found in the previous analysis). In practice, since the number of communities a VP observes depends on the filtering by the ASes in its vicinity, analysis requiring either diversity of BGP communities or communities from a specific AS requires a careful choice of VPs/collectors. Figure 6.5(d)[5], shows the VPs as circles (inner colored circles) with diameter and color proportional to the number of distinct AS identifiers (inferred from the two most-significant bytes of the community value) in the BGP communities they observe. The aggregated data (by collector and by data collection project) are depicted as grey circles, and highlight which collectors observe a more heterogeneous set of BGP communities: RouteViews collectors *route-views2* (3,624), *linx* (3,262), *route-views4* (3,236), and RIPE RIS collectors *rrc04* (2,979), *rrc01* (2,947), and *rrc12* (2,886). We selected the two collectors used for the analysis in Section 6.4.3 based on these data.

Performing analyses such as those discussed in this section without using BGP-Stream requires considering scalability issues (besides the efforts described in Section 6.4.2: crawling, data indexing, ASCII output parsing). For example, the amount of data needed for large longitudinal analyses may preclude a-priori download. In this case, a researcher would need to develop a system to dynamically download a moving window of data for consumption by analysis code. However, such a solution will turn storage into a potential bottleneck, since the size of the window limits the number of processing units that can run in parallel. A better solution would instead enable processing scripts to download data on demand, which is close (but still suboptimal) to the functionality provided by libBGPStream (which does not download the file to disk but streams it to the script directly from the HTTP connection). Another benefit of such functionality in a cluster-computing context is that it reduces the overhead of data locality optimization, since it implicitly co-locates each data block with the appropriate processor.

[5]An interactive, high-resolution version of this graph, as well as the equivalent for IPv6, are available at [178].

6.6 Continuous Monitoring

6.6.1 Lightweight monitoring: BGPCorsaro

BGPCorsaro is a tool to continuously extract derived data from a BGP stream in regular time bins. Its architecture is based on a pipeline of plugins, which continuously process BGPStream records. Plugins can be either:

- Stateless: e.g., performing classification and tagging of BGP records; plugins following in the pipeline can use such tags to inform their processing.
- Stateful: e.g., extracting statistics or aggregating data that are output at the end of each time bin. Since libBGPStream provides a sorted stream of records, BGPCorsaro can easily recognize the end of a time bin even when processing data from multiple collectors.

Both the core and the plugins of BGPCorsaro are written in C in order to support high-speed analysis of historical or live data streams. In Section 6.6.2, we describe a deployment of BGPCorsaro that runs 24/7 as a part of our global Internet monitoring infrastructure.

As a sample plugin, we describe a stateful plugin that monitors prefixes overlapping with a given set of IP address ranges. For each BGPStream record, the plugin: (1) selects only the RIB and Updates dump records related to prefixes that overlap with the given IP address ranges. (2) tracks, for each <prefix, VP> pair, the ASN that originated the route to the prefix. At the end of each time bin, the plugin outputs the timestamp of the current bin, the number of unique prefixes identified and, the number of unique origin ASNs observed by all the VPs.

we use a BGP hijacking event reported by Dyn Research, the hijacking of Italian Academic and Research Network (GARR) prefixes on January 7th 2015 [181], to demonstrate this plugin. we configured the plugin to process data from all available RouteViews and RIPE RIS collectors for January 2015, setting the time bin size to 5 minutes, and providing as input to the plugin the IP ranges covered by the 78 prefixes originated by AS137 (GARR) as observed on January 1st, 2015. Figure 6.6 shows a graphical representation of the two time-series generated by the plugin: the number of unique announced prefixes (in green) and number of unique origin ASNs (in blue). While a small oscillation of the number of prefixes announced is expected (as prefixes

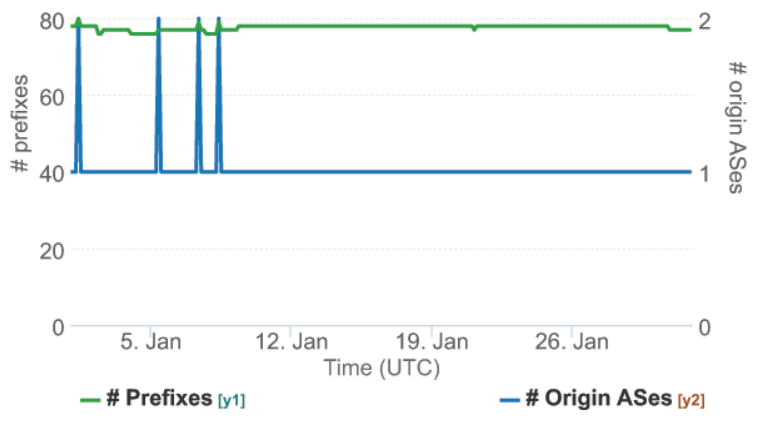


Fig. 6.6 Monitoring of GARR (AS137) IP space using the pfxmonitor plugin.

can be announced as aggregated or de-aggregated), in 4 cases the number of unique announcing ASes shifts from 1 to 2, for about 1 hour. Through manual analysis, we found that during these spikes a portion of GARR's IP space (specifically, 7 /24 prefixes) was also announced by TehnoGrup (AS 198596), a Romanian AS that appears to have no relationship with GARR. The report by Dyn Research describes a single attack on January 7th. However, given the similar nature of the other three events visible in the graph (1st, 5th and 8th of January), the plugin output suggests that three additional attacks occurred. Although this approach cannot detect all types of hijacking attacks, it is still a valid method to identify suspicious events and serves to demonstrate how users can leverage the capabilities of BGPCorsaro by writing plugins specific to their application.

6.6.2 Monitoring the Global Internet

In this section, we present a distributed architecture built on top of BGPStream and leveraging Apache Kafka [182] (a distributed messaging system) to perform continuous global BGP monitoring. My goal is two-fold: we demonstrate how BGPStream enables and simplifies developing complex global monitoring infrastructure and we present our architectural solutions to challenges that arise in this context.

To frame context and motivation for developing such complex architectures, let us consider two sample applications, our "Internet Outages: Detection and Analysis" (IODA) [183] and "Hijacks" [159] research projects. In IODA we monitor the Internet 24/7 to detect and characterize phenomena of macroscopic connectivity

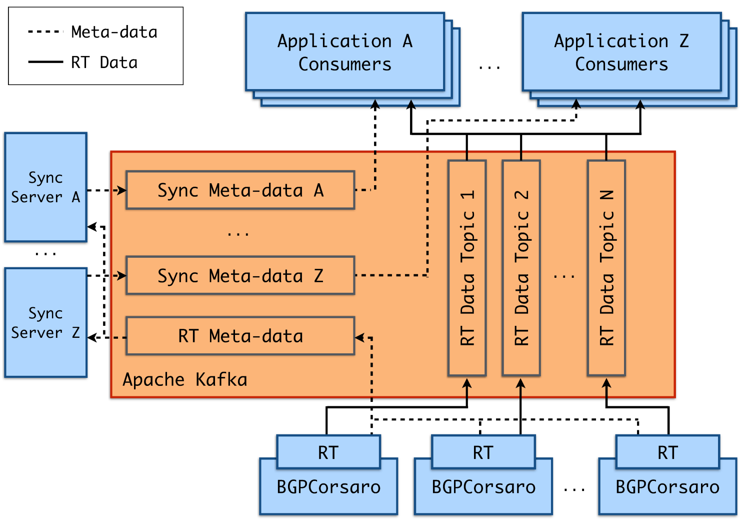


Fig. 6.7 Distributed framework for live monitoring.

disruption [184] [185]. In the case of BGP, our objective is to understand whether a set of prefixes (e.g., that share the same geographical region, or the same origin AS) are globally reachable or not. Information from a single VP is not sufficient to verify the occurrence of an outage, in fact, a prefix may be not reachable from the VP because of a local routing failure. On the other hand, if several VPs, topologically and geographically dispersed, simultaneously lose visibility of a prefix, then the prefix itself is likely undergoing an outage. In Hijacks, we are interested in detecting and analyzing BGP-based traffic hijacking. Since most common hijacks manifest as two or more ASes announcing exactly the same prefix, or a portion of the same address space at the same time, detecting them requires comparing the prefix reachability information as observed from multiple VPs.

In order to detect these events in a timely fashion, we need to maintain a global (i.e., for each and every VP) view of BGP reachability information updated with fine time granularity (e.g., few minutes). Such a continuously updated global view can be useful in many other applications, such as tracking AS paths containing a particular AS, verifying the occurrence of a route leak, spotting new (suspicious) AS links appearing in the AS-graph, etc.

we sketch our proposed architecture in Figure 6.7: multiple BGPCorsaro process data by using the RT plugin which reconstructs the observable LocRIB of all the collector's VPs (one instance per collector, in order to distribute the computation across multiple CPUs/hosts), their output is stored into an Apache Kafka cluster and further processed by applications (*consumers*) based on meta-data generated by *synchronization servers*. In the following sections, we describe the main components of this architecture and which challenges they address: Section 6.6.2 explains how we efficiently and accurately reconstruct the observable LocRIB of each VP; Section 6.6.2 illustrates our solution to reduce the amount of data we store and later process with the consumers; Section 6.6.2 shows how we solve the problem of supporting different synchronization mechanisms based on the application requirements; finally, in Section 6.6.2 we provide an example of applications implemented as a consumer.

Reconstructing VPs routing tables

RIB dumps are typically available every 2 or 8 hmys. My goal is to reconstruct snapshots of the observable LocRIB (herein referred to as the *routing table*) of each VP with a granularity of 1 or few minutes. For this purpose, we developed a BGPCorsaro plugin, called *routing-tables (RT)*. The RT plugin uses a RIB dump as a starting reference and then relies on the Updates dumps to reconstruct the evolution of the routing table, using subsequent RIB dumps for sanity checking and correction. However, since this is an inference process based on distributed collection of heterogeneous measurement data, multiple things can go wrong: BGP sessions going down, corrupted data, dump files published out of order, etc. We address this problem by maintaining a finite state machine and data structures that model the state of the VP, its routing table, and our confidence that the modeled data is accurate. In particular, we deal with the following fmy special events: ***E1.*** we ignore *all* records of a RIB dump if libBGPStream marks at least one of its records as corrupted. ***E2.*** Since records from a single RIB dump have timestamps often spanning several minutes *and* RIB and Update dumps may be published out of order, it is possible for the plugin to receive a RIB dump with some records that are older than the latest Update records applied by the plugin. To cope with this issue, We check each individual record of a RIB dump and only apply information from the record if the timestamp of the record is more recent than the timestamp of information already applied by the plugin. ***E3.*** Upon receiving a corrupted Updates dump record we stop applying Updates and wait for the next RIB dump. ***E4.*** We force state transitions upon receiving certain VP state messages (e.g., receipt of a state message with the Established code [127] triggers a transition to the UP state).

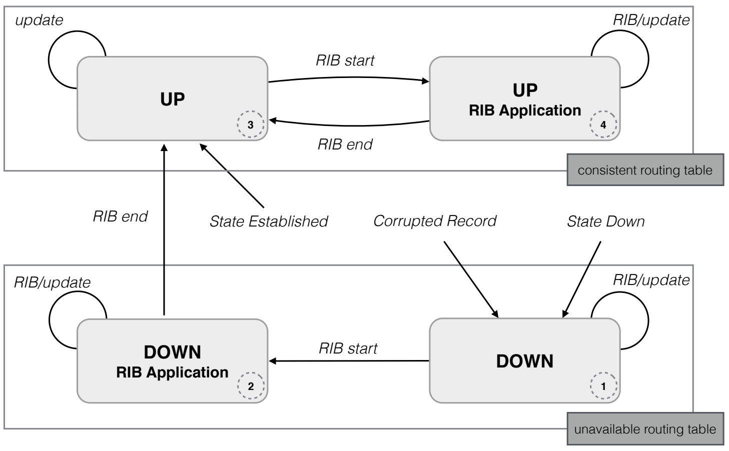


Fig. 6.8 Finite State Machine (FSM) for reconstructing VP routing table.

We save state and routing table information in a multi-dimensional hash table, which can be seen as a matrix with prefixes as rows and VPs as columns. Each cell contains the *reachability-attributes* for the prefix (e.g., the AS path), the *timestamp* of when the cell was last modified by an Updates dump record, and a *A/W* flag that indicates whether such operation was an announcement or a withdrawal. In addition, for each cell, the RT plugin uses a *shadow cell* to temporarily store records from a new RIB dump until it receives its last record: if none of the RIB dump records are corrupted (***E1***), we replace the content of the main cell with the content of the shadow cell unless the timestamp of the RIB record is older than the cell's last modification time (***E2***).

Figure 6.8 depicts the process of maintaining a VP routing table as a finite state machine that models the state of the VP. When the plugin starts, the VP's routing table is unavailable and the VP is in state *down* (1). When a new RIB dump starts, the VP's state moves to *down-RIB-application* state (2). During this phase, the plugin populates the shadow cells with the information received from the RIB dump records and the main cells with Updates dump records. The VP's state becomes *up* (3) once the entire RIB dump is received; when in this state the routing table is determined to be an accurate representation of the VP's routing table. Each new announcement or withdrawal record triggers modification of the main cell, whereas if a new RIB dump starts, the VP's state transitions to *up-RIB-application* (4), a state similar to (3) but whereby the RIB dump records modify the shadow information of the cells. Once the RIB ends, the shadow and main cells are merged (as described previously) and the VP transitions to state (3) again. In addition, a corrupted Updates dump record forces the state to be *down* (***E3***). Reception of an Updates dump record carrying a state message[6] with the Established code [127] moves the VP's state to *up* (***E4***), whereas reception of any other state message indicates that the connection between the VP and the collector is not established, and therefore, the VP is considered *down* (***E4***).

To evaluate the accuracy of our approach, we periodically compare the information in the current and shadow cell. RIS and RouteViews error probabilities – defined as the number of mismatching prefixes over the sum of all VPs' prefixes – calculated over 12 months across 31 collectors, are 10^{-8} and 10^{-5} respectively. We find that mismatches are usually caused by unresponsive VPs for which we do not have state messages (e.g., RouteViews), or by a collector not applying all incoming update messages before starting its RIB dump (but applying them afterwards, even if they have been already assigned a timestamp).

IO routines: diffs, (de)serialization, Kafka

At the end of each time bin, the RT plugin transmits the reconstructed routing table of each VP to a Kafka cluster. However, in order to reduce the volume of data to be stored and later processed by the consumers, we developed routines that allow the RT plugin to compute the difference between the routing table generated at the previous time bin and the current one and transmit only the changed portions (which we call *diff cells*). Consumers use complementary routines to retrieve the data from Kafka and reconstruct a full routing table by applying diffs to the previously stored version. The resulting data structure marks the updated portions of the routing table, allowing a consumer to limit its analysis to only these data. We periodically (e.g., 1 hmy) also store entire (non-diff) routing tables in the Kafka cluster that applications can use for synchronizing in order to receive future diffs.

Figure 6.9 highlights the advantage (in terms of number of processed BGP elems) of processing only diffs between routing tables instead of processing every update message. we run the RT plugin on data from *route-views2* for the month of

[6]Each RIPE collector maintains, for each VP, a finite state machine for the status of the BGP session with the VP and dump specific messages when state transitions occur. RouteViews collectors do not dump such state messages, hence the plugin may maintain a stale routing table for a VP that is actually down. To mitigate this problem, we also declare a VP down if none of its routes are present in the latest RIB dump.

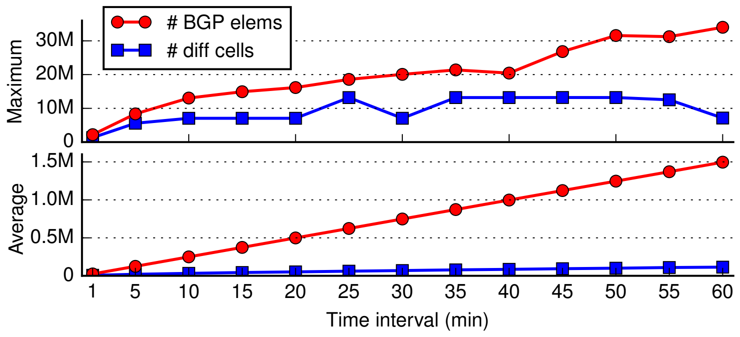


Fig. 6.9 RT diffs vs. BGP elems.

March 2016: in the graph, the red circles show the average (bottom) and maximum (top) number of BGP elems extracted from BGP update messages in each time bin, whereas the blue squares show the number of *diff cells* between consecutive routing tables. When the time bin is 1 minute, there are on average more than 3 times fewer diff cells than BGP elems, indicating that there is redundancy in update messages even at such short time scales. As the size of the time bin increases, the reduction factor also increases, at the expense of time granularity; a time bin of 1 hmy yields ≈13 times fewer diff cells than BGP elems. Also, the maxima show that by processing diffs, consumers are more resilient to bursts of updates (e.g., as a result of prefixes flapping).

Data synchronization

Different collectors, and in general different data smyces, provide data with variable delay. Performing data synchronization requires a trade-off between latency, amount of data available at processing time, and memory footprint. The optimal point in such a trade-off depends on the specific application goals and requirements. A monitoring application may require data from all (or a given fraction of) available smyces for the current time bin regardless of latency. Other applications may have stringent real-time requirements and prefer to explicitly set a time-out. For example: in realtime detection of hijacking, we set a time-out of few minutes to execute traceroutes as soon as a suspicious BGP event is detected; in the IODA application instead, we relax latency constraints in favor of data completeness and we use a time-out of 30 minutes, since it results in RT routing tables from all the VPs to be available for consumption for 99% of the time bins (we verified it on data from 2014 and 2015).

We designed a system based on meta-data stored in Kafka and multiple *sync servers*, each implementing a different synchronization mechanism: each BGP Corsaro RT plugin writes in the Kafka queue, along with the routing tables, indexing meta-data; such meta-data is monitored by the sync servers, which based on the synchronization criterion they implement, inject meta-data into their own topic in the Kafka queue to mark data as ready for consumption. By using Kafka, the resulting system is horizontally scalable (since Kafka supports distributing data across many nodes) and robust (e.g., due to data replication). In addition, since sync servers only handle lightweight meta-data which have a small memory footprint, they do not affect scalability.

Consumers

Consumers implement routines that analyze the routing tables retrieved from Kafka to perform event detection, extraction of statistics to output as time series etc. We developed two consumers for near-realtime detection of per-country and per-AS outages. Both consumers select the prefixes observed by full-feed VPs and monitor the visibility of these prefixes by computing the number of prefixes geo-located to each country and announced by each AS. The consumers store this data into a time series monitoring system supporting automated change-point detection and data visualization.

Figure 6.10 shows data from the per-country and per-AS outages consumers over a period of 1 month, (June 20 to July 20, 2015), selecting prefix visibility associated with Iraq and five of the biggest Iraqi ISPs. The noticeable drops reflect a sequence of country-wide Internet outages that the government ordered in conjunction with the ministerial preparatory exams [186–188].

6.7 Conclusions

BGPStream targets a broad range of applications and users. We hope that it will enable novel analyses, development of new tools, educational opportunities, as well as feedback and contributions to our platform. In addition, since code and scripts using BGPStream embed the definition of the public data sources used for

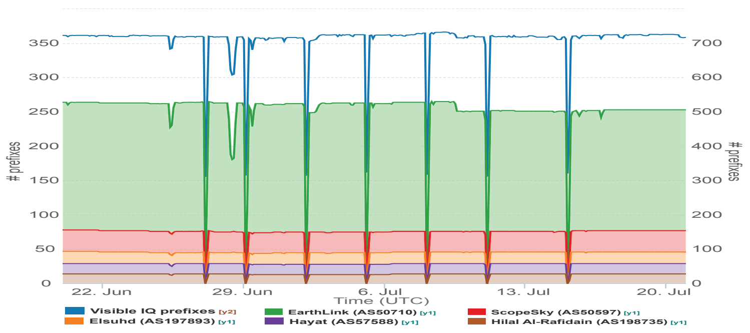


Fig. 6.10 Visible Iraqi prefixes (June, 20- July, 20 2015).

an experiment, BGPStream significantly eases the reproducibility of experimental results.

BGPStream development is part of a collaborative effort with other researchers and data providers, such as Cisco, RouteViews and BGPMon, to coordinate progress in this space [156]. We plan to release new features in the near future, including support for more data formats (e.g., JSON exports from ExaBGP [189], OpenBMP [128]). In particular, adding native support for OpenBMP will enable processing of streams sourced directly from BGP routers.

Chapter 7

Conclusions and Discussion

In this thesis I presented different studies to show how network monitoring is a fundamental task to gain knowledge about the network. I showed that all the the studies I proposed follow a common workflow starting with the data acquisition phase through, *active* or *passive* monitoring. In Cap. 2, Cap. 3, and Cap. 5 I showed how to take advantage of passive monitoring by analyzing data acquired with a passive probe. In Cap. 4 and Cap. 6, I showed the benefits of using active and passive monitoring in synergy. After that all the proposed studies continue with a *feature selection* phase driven by my domain knowledge. Then, each chapter, presented a specific study which falls in one or more of four categories: server monitoring (Cap. 2 and Cap. 4), client monitoring (Cap. 3), network monitoring (Cap. 6), or the application of novel methodologies (Cap. 2, Cap. 4, and Cap. 6).

This thesis offered a walk through methodologies from the mathematical, statistical and set theory worlds (Cap. 3 and Cap. 4), to the machine learning community (Cap. 2, and Cap. 5), finishing with big data approaches (Cap. 5, and Cap. 6). All these studies have the aim to demonstrate how machine learning and big data methodologies are nowadays fundamental to automatically extract meaningful information from a dataset.

In particular, in Cap. 2 with *YouLighter* I showed how to monitor the YouTube CDN by using passive monitoring. *YouLighter* uses DBSCAN, an unsupervised machine learning methodology, to automatically group thousands of YouTube caches into few edge-nodes. Then, by exploiting the notion of *Pattern Dissimilarity* I showed how to compare two clustering results to evaluate their evolution of over time. In particular, *YouLighter* has been able to sport a large transformation in a crucial edge-node of YouTube CDN, that also impaired the QoE perceived by the monitored ISP customers for more than 40 days.

Instead in Cap. 3, by using a dataset from passive monitoring and by investigating the browsing habits of Internauts, I showed that use the DNS traffic to build a users' profiler is not effective. The reason is that the similarity of the browsing activity during two different periods of time is rather limited. In the one head, this is due to the presence of a lot of websites that are contacted by the users without their knowledge. In the other hand, because we repetitively visit only a small subset of all the websites that we contact every day. Indeed, in my experiment only 44% of the time I was able to uniquely identify users based on their activity. With this percentage reducing over time.

In Cap. 4, by exploiting a census of anycast addresses made by active monitoring, I analyzed the pervasiveness of Anycast-enabled CDNs (A-CDNs) in a dataset obtained through passive monitoring. I demonstrated that: A-CDNs are a reality as they are visited my more then 50% of web users every day, and they are used by several important players for load balancing. I showed that they are used to offer either stateless services such as web pages e.g., WordPress, either stateful services, such as audio/video streaming e.g., Soundcloud, or Vine. The possibility to apply anycast addressing also to stateful services is guarantee by the stability of A-CDNs paths. The prof, is that by analyzing all A-CDNs RTT over the whole dataset, I discovered only a few anomalies in the RTT to reach A-CDNs.

To demonstrate the possibility to use a machine learning solutions to automatically spot changes, in cap. 5, I presented SeLINA a self-tuning tool implemented in the big data Apache Spark big data framework. I evaluated SeLINA analyzing with two datasets obtained from passive monitoring. In the first dataset, SeLINA demonstrated to be effective in the automatic detection of changes as it as been able to automatically pinpoint the YouTube anomaly previously highlighted with *YouLighter*. In the second dataset, by analyzing P2P traffic, SeLINA highlighted some interesting clusters, but no anomalies over time.

To conclude, Cap. 6 presented BGPStream, an open-source software framework for the analysis of both historical and real-time Border Gateway Protocol (BGP) measurement data. BGPStream features demonstrated to be very useful in different scenario. First, in real-time and using only passive data, BGPStream highlighted a well known MOAS attack at the GARR prefix space. Then, in cooperation with active monitoring, BGPStream allowed me to evaluate the propagation of the blackholing communities in the network. Finally, by using the historical analysis features, BGPStream allowed me to analyze over 16 years of measurement data exploiting the Apache Spark big data framework. With this framework I measured different aspects such as the growth of the Internet in term of IPv4 prefixes, or the number of MOAS events.

Despite this thesis presented different monitoring solutions in various scenarios, they represent only a small subset of possible application of machine learning methodologies in the network monitoring field. Each methodology demonstrated to be effective in a particular scenario, and I hope that my solutions can be a starting point to design new methodologies by other researchers. The application of big data tools has been briefly touched in this thesis. However, I strongly believe that these tools will be the future for the network monitoring problem given the rise of networking data. One last but not least important aspect to keep in mind while monitoring is the users' privacy. In my thesis I always used anonymized or aggregate data for this reason. Often, while we monitor the network, we acquire and study data containing sensitive users' information. For this reason, before performing any analysis, and before evaluating the technical aspects, it is important to the evaluate ethics implications of each study, in order to protect the users' privacy and the inappropriate usage of their data.

References

[1] Cisco. The Zettabyte Era — Trends and Analysis – Cisco (technical report). http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/vni-hyperconnectivity-wp.pdf, 2016.

[2] INTERNET GROWTH STATISTICS. http://www.internetworldstats.com/emarketing.htm, 2016.

[3] Pedro Casas, Alessandro D'Alconzo, Pierdomenico Fiadino, Arian Bär, Alessandro Finamore, and Tanja Zseby. When youtube does not work - analysis of qoe-relevant degradation in google CDN traffic. *IEEE Transactions on Network and Service Management*, 11(4):441–457, 2014.

[4] Arian Bär, Alessandro Finamore, Pedro Casas, Lukasz Golab, and Marco Mellia. Large-scale network traffic monitoring with dbstream, a system for rolling big data analysis. In *2014 IEEE International Conference on Big Data, Big Data 2014, Washington, DC, USA, October 27-30, 2014*, pages 165–170, 2014.

[5] Pedro Casas, Johan Mazel, and Philippe Owezarski. Unsupervised network intrusion detection systems: Detecting the unknown without knowledge. *Computer Communications*, 35(7):772–783, 2012.

[6] Juliette Dromard, Gilles Roudiere, and Philippe Owezarski. Unsupervised network anomaly detection in real-time on big data. In *New Trends in Databases and Information Systems - ADBIS 2015 Short Papers and Workshops, BigDap, DCSA, GID, MEBIS, OAIS, SW4CH, WISARD, Poitiers, France, September 8-11, 2015. Proceedings*, pages 197–206, 2015.

[7] Kuai Xu, Zhi-Li Zhang, and Supratik Bhattacharyya. Internet traffic behavior profiling for network security monitoring. *IEEE/ACM Transactions On Networking*, 16(6):1241–1252, 2008.

[8] Q Guo, W Wu, D.L Massart, C Boucon, and S de Jong. Feature selection in principal component analysis of analytical data. *Chemometrics and Intelligent Laboratory Systems*, 61(1–2):123 – 132, 2002.

[9] Hanchuan Peng, Fuhui Long, and C. Ding. Feature selection based on mutual information criteria of max-dependency, max-relevance, and min-redundancy.

IEEE Transactions on Pattern Analysis and Machine Intelligence, 27(8):1226–1238, Aug 2005.

[10] D. Bonfiglio, M. Mellia, M. Meo, N. Ritacca, and D. Rossi. Tracking down skype traffic. In *IEEE INFOCOM 2008 - The 27th Conference on Computer Communications*, April 2008.

[11] Ye Ouyang, M. Hosein Fallah, Sanqing Hu, Yong Ren Yong, Yirui Hu, Zhichang Lai, Mingxin Guan, and Wenyuan Lu. A novel methodology of data analytics and modeling to evaluate LTE network performance. In *2014 Wireless Telecommunications Symposium, WTS 2014, Washington, DC, USA, April 9-11, 2014*, pages 1–10, 2014.

[12] Michael J Berry and Gordon Linoff. *Data mining techniques: for marketing, sales, and customer support*. John Wiley & Sons, Inc., 1997.

[13] A Soltani Sarvestani, AA Safavi, NM Parandeh, and M Salehi. Predicting breast cancer survivability using data mining techniques. In *Software technology and Engineering (ICSTE), 2010 2nd international Conference on*, volume 2, pages V2–227. IEEE, 2010.

[14] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *2010 IEEE Symposium on Security and Privacy*, pages 305–316, May 2010.

[15] Konstantin Tretyakov. Machine learning techniques in spam filtering. In *Data Mining Problem-oriented Seminar, MTAT*, volume 3, pages 60–79, 2004.

[16] Peter Burge and John Shawe-Taylor. An unsupervised neural network approach to profiling the behavior of mobile phone users for use in fraud detection. *Journal of parallel and distributed computing*, 61(7):915–925, 2001.

[17] Gerhard Münz, Sa Li, and Georg Carle. Traffic anomaly detection using k-means clustering. In *GI/ITG Workshop MMBnet*, 2007.

[18] Ruben D. Torres, Mohammad Y. Hajjat, Sanjay G. Rao, Marco Mellia, and Maurizio M. Munafo. Inferring undesirable behavior from p2p traffic analysis. In *SIGMETRICS Performance Evaluation Review*, volume 37, pages 25 – 36. ACM, 2009.

[19] Alan Bivens, Chandrika Palagiri, Rasheda Smith, Boleslaw Szymanski, Mark Embrechts, et al. Network-based intrusion detection using neural networks. *Intelligent Engineering Systems through Artificial Neural Networks*, 12(1):579–584, 2002.

[20] Robert Cooley, Bamshad Mobasher, and Jaideep Srivastava. Web mining: Information and pattern discovery on the world wide web. In *Tools with Artificial Intelligence, 1997. Proceedings., Ninth IEEE International Conference on*, pages 558–567. IEEE, 1997.

[21] Apache Hadoop. The Apache The Apache Hadoop project. Available: http://hadoop.apache.org. 2015.

[22] Apache Spark. http://spark.apache.org/, 2015.

[23] Alessandro Finamore, Marco Mellia, Michela Meo, Maurizio M. Munafò, and Dario Rossi. Experiences of Internet Traffic Monitoring with Tstat. *IEEE Network*, 25:8–14, 2011.

[24] Alessandro Finamore, Marco Mellia, Michela Meo, Maurizio M Munafo, and Dario Rossi. Experiences of internet traffic monitoring with tstat. *Network, IEEE*, 25(3):8–14, 2011.

[25] Dario Rossi and Marco Mellia. Real-time tcp/ip analysis with common hardware. In *Communications, 2006. ICC'06. IEEE International Conference on*, volume 2, pages 729–735. IEEE, 2006.

[26] Martino Trevisan, Finamore Alessandro, Marco Mellia, Maurizio Munafò, and Dario Rossi. DPDKStat: 40Gbps Statistical Traffic Analysis with Off-the-Shelf Hardware. *In Tech. Rep., 2016.*

[27] Ignacio Bermudez, Marco Mellia, Maurizio Munafò, Ram Keralapura, and Antonio Nucci. DNS to the Rescue: Discerning Content and Services in a Tangled Web. ACM IMC, 2012.

[28] Matt Calder, Xun Fan, Zi Hu, Ethan Katz-Bassett, John Heidemann, and Ramesh Govindan. Mapping the expansion of google's serving infrastructure. In *ACM IMC*, 2013.

[29] V.K. Adhikari, S. Jain, Yingying Chen, and Zhi-Li Zhang. Vivisecting youtube: An active measurement study. In *IEEE INFOCOM*, 2012.

[30] R. Torres, A. Finamore, Jin Ryong Kim, M. Mellia, M.M. Munafo, and Sanjay Rao. Dissecting video server selection strategies in the youtube cdn. In *IEEE ICDCS*, 2011.

[31] P. Casas, Pierdomenico Fiadino, and A Bär. Understanding http traffic and cdn behavior from the eyes of a mobile isp. In *PAM*, 2014.

[32] P. Casas, A. D'Alconzo, P. Fiadino, A. Bar, A. Finamore, and T. Zseby. When youtube does not work: Analysis of qoe-relevant degradation in google cdn traffic. *Network and Service Management, IEEE Transactions on*, 11(4):441–457, Dec 2014.

[33] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *ACM KDD*, 1996.

[34] Tobias Hossfeld, Raimund Schatz, Ernst Biersack, and Louis Plissonneau. Internet video delivery in youtube: From traffic measurements to quality of experience. In *Data Traffic Monitoring and Analysis*, volume 7754, pages 264–301. Springer, 2013.

[35] Animesh Patcha and Jung-Min Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Comput. Netw.*, 51(12):3448–3470, August 2007.

[36] Varun Chandola, Arindam Banerjee, and Vipin Kumar. Anomaly detection: A survey. In *Computing Surveys*, volume 41, pages 1 – 58. ACM, 2009.

[37] He Yan, A Flavel, Zihui Ge, A Gerber, D. Massey, C. Papadopoulos, H. Shah, and J. Yates. Argus: End-to-end service anomaly detection and localization from an isp's point of view. In *IEEE INFOCOM*, 2012.

[38] Anukool Lakhina, Mark Crovella, and Christophe Diot. Mining anomalies using traffic feature distributions. *SIGCOMM Comput. Commun. Rev.*, 35(4):217–228, August 2005.

[39] Junchen Jiang, Vyas Sekar, Ion Stoica, and Hui Zhang. Shedding light on the structure of internet video quality problems in the wild. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 357–368, New York, NY, USA, 2013. ACM.

[40] Pierdomenico Fiadino, Alessandro D'Alconzo, Arian Bar, Alessandro Finamore, and Pedro Casas. On the detection of network traffic anomalies in content delivery network services. In *Teletraffic Congress (ITC), 2014 26th International*, pages 1–9. IEEE, 2014.

[41] Deepayan Chakrabarti, Ravi Kumar, and Andrew Tomkins. Evolutionary clustering. In *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, pages 554–560, New York, NY, USA, 2006. ACM.

[42] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003.

[43] Mark K Goldberg, Mykola Hayvanovych, and Malik Magdon-Ismail. Measuring similarity between sets of overlapping clusters. In *IEEE SocialCom*, 2010.

[44] Panos Kalnis, Nikos Mamoulis, and Spiridon Bakiras. On discovering moving clusters in spatio-temporal data. In *Advances in spatial and temporal databases*, volume 37, pages 364 – 381. Springer, 2005.

[45] Zhenhui Li, Bolin Ding, Jiawei Han, and Roland Kays. Swarm: Mining relaxed temporal moving object clusters. In *Proceedings of the VLDB Endowment*, volume 3, pages 723–734. VLDB Endowment, 2010.

[46] Daniel Kifer, Shai Ben-David, and Johannes Gehrke. Detecting change in data streams. In *ACM VLDB*, 2004.

[47] D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar, and S. Saha. Youlighter: An unsupervised methodology to unveil youtube cdn changes. In *27th International Teletraffic Congress*, 2015.

[48] David Naylor, Alessandro Finamore, Ilias Leontiadis, Yan Grunenberger, Marco Mellia, Maurizio Munafò, Konstantina Papagiannaki, and Peter Steenkiste. The cost of the s in https. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 133–140. ACM, 2014.

[49] Ignacio N. Bermudez, Marco Mellia, Maurizio M. Munafo, Ram Keralapura, and Antonio Nucci. Dns to the rescue: Discerning content and services in a tangled web. In ACM, editor, *IMC*, 2012.

[50] Pang-Ning Tan, Michael Steinbach, Vipin Kumar, et al. *Introduction to data mining*, volume 1. Pearson Addison Wesley Boston, 2006.

[51] P. Casas, A. Sackl, S. Egger, and R. Schatz. Youtube amp; facebook quality of experience in mobile broadband networks. In *Globecom Workshops (GC Wkshps), 2012 IEEE*, pages 1269–1274, Dec 2012.

[52] Bruno Quoitin, Cristel Pelsser, Louis Swinnen, Ouvier Bonaventure, and Steve Uhlig. Interdomain traffic engineering with bgp. *Communications Magazine, IEEE*, 41(5):122–128, 2003.

[53] Lara D Catledge and James E Pitkow. Characterizing browsing strategies in the world-wide web. *Computer Networks and ISDN systems*, 27(6):1065–1073, 1995.

[54] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. Youtube traffic characterization: a view from the edge. In *ACM IMC*, 2009.

[55] Alessandro Finamore, Marco Mellia, Maurizio M Munafò, Ruben Torres, and Sanjay G Rao. Youtube everywhere: Impact of device and infrastructure synergies on user experience. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 345–360. ACM, 2011.

[56] Chad Tossell, Philip Kortum, Ahmad Rahmati, Clayton Shepard, and Lin Zhong. Characterizing web use on smartphones. In *ACM SIGCHI*, 2012.

[57] Daniel Olmedilla, Enrique Frías-Martínez, and Rubén Lara. Mobile web profiling: A study of off-portal surfing habits of mobile users. In *User Modeling, Adaptation, and Personalization*, volume 6075 of *Lecture Notes in Computer Science*, pages 339–350. Springer Berlin Heidelberg, 2010.

[58] Fabrício Benevenuto, Tiago Rodrigues, Meeyoung Cha, and Virgílio Almeida. Characterizing user behavior in online social networks. In *ACM IMC*, 2009.

[59] Network Working Group et al. Request for comments (rfc) 4033,“. *Protocol Modifications for the DNS Security Extensions,” Mar*, 2005.

[60] https://www.opendns.com/about/innovations/dnscrypt/.

[61] Bernhard Ager, Holger Dreger, and Anja Feldmann. Predicting the dnssec overhead using dns traces. In *IEEE CISS*, 2006.

[62] Marek Kumpošt and Vašek Matyáš. *User Profiling and Re-identification: Case of University-Wide Network Analysis*. Springer, 2009.

[63] Lukasz Olejnik, Claude Castelluccia, and Artur Janc. Why johnny can’t browse in peace: On the uniqueness of web browsing history patterns. In *HotPETs*, 2012.

[64] Dominik Herrmann, Christian Banse, and Hannes Federrath. Behavior-based tracking: Exploiting characteristic patterns in dns traffic. *Computers & Security*, 39:17–33, 2013.

[65] Ram Keralapura, Antonio Nucci, Zhi-Li Zhang, and Lixin Gao. Profiling users in a 3g network using hourglass co-clustering. In *ACM MobiCom*, 2010.

[66] Vinicius Gehlen, Alessandro Finamore, Marco Mellia, and Maurizio M Munafo. *Uncovering the big players of the web*. 2012.

[67] Michael Levandowsky and David Winter. Distance between sets. *Nature*, 234(5323):34–35, 1971.

[68] Lada A Adamic and Bernardo A Huberman. Zipf’s law and the internet. *Glottometrics*, 3(1):143–150, 2002.

[69] Ted Hardie. Known Content Network (CN) Request-Routing Mechanisms. IETF RFC 3568, 2003.

[70] Zakaria Al-Qudah, Seungjoon Lee, Michael Rabinovich, Oliver Spatscheck, and Jacobus Van der Merwe. Anycast-aware Transport for Content Delivery Networks. In *Proc. ACM WWW*, 2009.

[71] Hussein A. Alzoubi, Seungjoon Lee, Michael Rabinovich, Oliver Spatscheck, and Jacobus Van Der Merwe. A practical architecture for an anycast cdn. *ACM Trans. Web*, 5(4):17:1–17:29, Oct 2011.

[72] Ashley Flavel, Pradeepkumar Mani, David A Maltz, Nick Holt, Jie Liu, Yingying Chen, and Oleg Surmachev. FastRoute: A Scalable Load-Aware Anycast Routing Architecture for Modern CDNs. In *Proc. USENIX NSDI*, 2015.

[73] Danilo Cicalese, Jordan Augé, Diana Joumblatt, Timur Friedman, and Dario Rossi. Characterizing IPv4 Anycast Adoption and Deployment. In *Proc. CoNEXT*, 2015.

[74] Dina Katabi and John Wroclawski. A Framework for Scalable Global IP-anycast (GIA). In *Proc. ACM SIGCOMM*, 2000.

[75] Matt Calder, Ashley Flavel, Ethan Katz-Bassett, Ratul Mahajan, and Jitendra Padhye. Analyzing the Performance of an Anycast CDN. In *Proc. ACM IMC*, 2015.

[76] Matt Levine, Barrett Lyon, and Todd Underwood. Operational Experience with TCP and Anycast. Nanog, 2006.

[77] Ziqian Liu, Bradley Huffaker, Marina Fomenkov, Nevil Brownlee, and Kimberly C. Claffy. Two Days in the Life of the DNS Anycast Root Servers. In *Proc. of PAM*, 2007.

[78] Sandeep Sarat, Vasileios Pappas, and Andreas Terzis. On the Use of Anycast in DNS. In *Proc. IEEE ICCCN*, 2006.

[79] Hitesh Ballani, Paul Francis, and Sylvia Ratnasamy. A Measurement-based Deployment Proposal for IP Anycast. In *Proc. ACM IMC*, 2006.

[80] Lorenzo Colitti. Measuring Anycast Server Performance: The Case of K-root. Nanog, 2006.

[81] Peter Boothe and Randy Bush. DNS Anycast Stability: Some Early Results. CAIDA, 2005.

[82] Biet Barber, Matt Larson, and Mark Kosters. Traffic Source Analysis of the J Root Anycast instances. Nanog, 2006.

[83] Daniel Karrenberg. Anycast and BGP Stability: A Closer Look at DNSMON Data. Nanog, 2005.

[84] Doug Madory, Chris Cook, and Kevin Miao. Who Are the Anycasters. Nanog, 2013.

[85] Xun Fan, John S. Heidemann, and Ramesh Govindan. Evaluating Anycast in the Domain Name System. In *Proc. IEEE INFOCOM*, 2013.

[86] Danilo Cicalese, Diana Joumblatt, Dario Rossi, Marc-Olivier Buob, Jordan Augé, and Timur Friedman. A Fistful of Pings: Accurate and Lightweight Anycast Enumeration and Geolocation. In *Proc. IEEE INFOCOM*, 2015.

[87] Danilo Cicalese, Danilo Giordano, Alessandro Finamore, Marco Mellia, Maurizio Munafò, Dario Rossi, and Diana Joumblatt. A First Look at Anycast CDN Traffic. *ArXiv e-prints*, 2016.

[88] D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar, and S. Saha. Youlighter: A cognitive approach to unveil youtube cdn and changes. *IEEE Transactions on Cognitive Communications and Networking*, 1(2):161–174, June 2015.

[89] Yeonhee Lee and Youngseok Lee. Toward scalable internet traffic measurement and analysis with hadoop. *ACM SIGCOMM Computer Communication Review*, 43(1):5–13, 2013.

[90] Daniele Apiletti, Elena Baralis, Tania Cerquitelli, and Vincenzo D'Elia. Characterizing network traffic by means of the netmine framework. *Computer Networks*, 53(6):774–789, 2009.

[91] M. Hossain, SM Bridges, and RB Vaughn Jr. Adaptive intrusion detection with data mining. *IEEE Internation Conference on Systems, Man and Cybernetics*, 4, 2003.

[92] Franck Le, Sihyung Lee, Tina Wong, Hyong S. Kim, and Darrell Newcomb. Minerals: using data mining to detect router misconfigurations. In *MineNet '06*, pages 293–298, New York, NY, USA, 2006. ACM Press.

[93] Manoj K. Agarwal, Manish Gupta, Gautam Kar, Anindya Neogi, and Anca Sailer. Mining activity data for dynamic dependency discovery in e-business systems. *IEEE Transactions on Network and Service Management*, 1(2):49–58, 2004.

[94] Thomas Karagiannis, Konstantina Papagiannaki, and Michalis Faloutsos. Blinc: multilevel traffic classification in the dark. In *SIGCOMM*, pages 229–240, 2005.

[95] Andrew W. Moore and Denis Zuev. Internet traffic classification using bayesian analysis techniques. In *SIGMETRICS '05*, pages 50–60, New York, NY, USA, 2005. ACM Press.

[96] José Everardo Bessa Maia et al. Network traffic prediction using pca and k-means. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 938–941. IEEE, 2010.

[97] Mennatallah Amer, Markus Goldstein, and Slim Abdennadher. Enhancing one-class support vector machines for unsupervised anomaly detection. In *Proceedings of the ACM SIGKDD Workshop on Outlier Detection and Description*, ODD '13, pages 8–15, 2013.

[98] L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. *Proceedings of ACM CSS Workshop on Data Mining Applied to Security, PA,, November*, 2001.

[99] Q. Wang and V. Megalooikonomu. A clustering algorithm for intrusion detection. *Proc. SPIE*, 5812:31–38, 2005.

[100] Philippe Owezarski. Unsupervised classification and characterization of honeypot attacks. In *10th International Conference on Network and Service Management, CNSM 2014 and Workshop, Rio de Janeiro, Brazil, November 17-21, 2014*, pages 10–18, 2014.

[101] Elena Baralis, Andrea Bianco, Tania Cerquitelli, Luca Chiaraviglio, and Marco Mellia. Netcluster: A clustering-based framework to analyze internet passive measurements data. *Computer Networks*, 57(17):3300–3315, 2013.

[102] Luigi Grimaudo, Marco Mellia, Elena Baralis, and Ram Keralapura. Select: Self-learning classifier for internet traffic. *IEEE Transactions on Network and Service Management*, 11(2):144–157, 2014.

[103] Jeffrey Erman, Martin Arlitt, and Anirban Mahanti. Traffic classification using clustering algorithms. In *MineNet '06*, pages 281–286, New York, NY, USA, 2006. ACM Press.

[104] Jao Yoon Chung, Byungchul Park, Young J Won, John Strassner, and James W Hong. An effective similarity metric for application traffic classification. In *Network Operations and Management Symposium (NOMS), 2010 IEEE*, pages 286–292. IEEE, 2010.

[105] Marcus Fabio Fontenelle do Carmo, Jose Everardo Bessa Maia, GP Siqueira, et al. An internet traffic classification methodology based on statistical discriminators. In *Network Operations and Management Symposium, 2008. NOMS 2008. IEEE*, pages 907–910. IEEE, 2008.

[106] F.A. Lisi and D. Malerba. Inducing multi-level association rules from multiple relations. *Machine Learning*, 55(2):175–210, 2004.

[107] Ricardo J. G. B. Campello, Davoud Moulavi, and Jörg Sander. Density-based clustering based on hierarchical density estimates. In *Advances in Knowledge Discovery and Data Mining, 17th Pacific-Asia Conference, PAKDD 2013, Gold Coast, Australia, April 14-17, 2013, Proceedings, Part II*, pages 160–172, 2013.

[108] Ricardo J. G. B. Campello, Davoud Moulavi, Arthur Zimek, and Jörg Sander. Hierarchical density estimates for data clustering, visualization, and outlier detection. *TKDD*, 10(1):5, 2015.

[109] Gregory Buehrer, Roberto L. de Oliveira Jr., David Fuhry, and Srinivasan Parthasarathy. Towards a parameter-free and parallel itemset mining algorithm in linearithmic time. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 1071–1082, 2015.

[110] Yaobin He, Haoyu Tan, Wuman Luo, Shengzhong Feng, and Jianping Fan. MR-DBSCAN: a scalable mapreduce-based DBSCAN algorithm for heavily skewed data. *Frontiers of Computer Science*, 8(1):83–99, 2014.

[111] Sandy Moens, Emin Aksehirli, and Bart Goethals. Frequent itemset mining for big data. In *Proceedings of the 2013 IEEE International Conference on Big Data, 6-9 October 2013, Santa Clara, CA, USA*, pages 111–118, 2013.

[112] Biswanath Panda, Joshua Herbach, Sugato Basu, and Roberto J. Bayardo. PLANET: massively parallel learning of tree ensembles with mapreduce. *PVLDB*, 2(2):1426–1437, 2009.

[113] Devendra Dahiphale, Rutvik Karve, Athanasios V. Vasilakos, Huan Liu, Zhiwei Yu, Amit Chhajer, Jianmin Wang, and Chaokun Wang. An advanced mapreduce: Cloud mapreduce, enhancements and applications. *IEEE Transactions on Network and Service Management*, 11(1):101–115, 2014.

[114] Apache Spark. The Apache Spark scalable machine learning library. Available: https://spark.apache.org/mllib/. 2015.

[115] Vernon KC Bumgardner and Victor W Marek. Scalable hybrid stream and hadoop network analysis system. In *Proceedings of the 5th ACM/SPEC international conference on Performance engineering*, pages 219–224. ACM, 2014.

[116] Yousun Jeong. Big Telco Real-Time Network Analytics Available: https://spark-summit.org/eu-2015/events/big-telco-real-time-network-analytics/. *Spark summit, Amsterdam, Netherland, October 27-29*, 2015.

[117] KV Swetha, Shiju Sathyadevan, and P Bilna. Network data analysis using spark. In *Software Engineering in Intelligent Systems*, pages 253–259. Springer, 2015.

[118] Daniele Apiletti, Elena Baralis, Tania Cerquitelli, Paolo Garza, and Luca Venturini. SaFe-NeC: a Scalable and Flexible system for Network data Characterization. In *NOMS*, 2016.

[119] Jiawei Han. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2005.

[120] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96), Portland, Oregon, USA*, pages 226–231, 1996.

[121] Dario Antonelli, Elena Baralis, Giulia Bruno, Tania Cerquitelli, Silvia Chiusano, and Naeem A. Mahoto. Analysis of diabetic patients through their examination history. *Expert Syst. Appl.*, 40(11):4672–4678, 2013.

[122] Lior Rokach and Oded Maimon. *Data Mining with Decision Trees: Theroy and Applications*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 2008.

[123] Leo Breiman, Jerome Friedman, Charles J Stone, and Richard A Olshen. *Classification and regression trees*. CRC press, 1984.

[124] Pang-Ning T. and Steinbach M. and Kumar V. *Introduction to Data Mining*. Addison-Wesley, 2006.

[125] Peter J. Rousseeuw. Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. *Journal of Computational and Applied Mathematics*, 20:53 – 65, 1987.

[126] J. L. Garcia-Dorado, A. Finamore, M. Mellia, M. Meo, and M. Munafo. Characterization of isp traffic: Trends, user habits, and access technology impact. *IEEE Transactions on Network and Service Management*, 9(2):142–155, June 2012.

[127] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271 (Draft Standard), Jan 2006. Updated by RFCs 6286, 6608, 6793, 7606, 7607.

[128] Tim Evens. OpenBMP. http://http://www.openbmp.org/, 2015.

[129] J. Scudder, R. Fernando, and S. Stuart. BGP Monitoring Protocol. Internet-Draft draft-ietf-grow-bmp-14.txt, IETF Secretariat, Aug 2015.

[130] University of Oregon. Route Views Project. http://www.routeviews.org/, 2015.

[131] RIPE NCC. Routing Information Service (RIS). https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris, 2015.

[132] PCH. Packet Clearing House. http://www.pch.net/, 2015.

[133] Quagga. Quagga Routing Software Suite. http://www.nongnu.org/quagga/, 2015.

[134] Matthew Luckie, Bradley Huffaker, Amogh Dhamdhere, Vasileios Giotsas, and k claffy. AS relationships, customer cones, and validation. In *IMC*, Oct 2013.

[135] Vasileios Giotsas, Matthew Luckie, Bradley Huffaker, et al. Inferring complex as relationships. In *Proceedings of the 2014 Conference on Internet Measurement Conference*, pages 23–30. ACM, 2014.

[136] Matthew Luckie. Spurious routes in public bgp data. *ACM SIGCOMM Computer Communication Review*, 44(3):14–21, 2014.

[137] Andra Lutu, Marcelo Bagnulo, Jesus Cid-Sueiro, and Olaf Maennel. Separating wheat from chaff: Winnowing unintended prefixes using machine learning. In *INFOCOM, 2014 Proceedings IEEE*, pages 943–951. IEEE, 2014.

[138] Matthias Wählisch, Olaf Maennel, and Thomas C Schmidt. Towards detecting bgp route hijacking using the rpki. *ACM SIGCOMM Computer Communication Review*, 42(4):103–104, 2012.

[139] Riad Mazloum, Marc-Olivier Buob, Jordan Auge, Bruno Baynat, Dario Rossi, and Timur Friedman. Violation of interdomain routing assumptions. In *Passive and Active Measurement*, pages 173–182. Springer, 2014.

[140] Quentin Jacquemart, Guillaume Urvoy-Keller, and Ernst Biersack. A longitudinal study of bgp moas prefixes. In *Traffic Monitoring and Analysis*, pages 127–138. Springer, 2014.

[141] Xin Hu and Z Morley Mao. Accurate real-time identification of ip prefix hijacking. In *Security and Privacy, 2007. SP'07. IEEE Symposium on*, pages 3–17. IEEE, 2007.

[142] L. Blunk, M. Karir, and C. Labovitz. Multi-Threaded Routing Toolkit (MRT) Routing Information Export Format. RFC 6396 (Proposed Standard), Oct 2011.

[143] Marijana Cosovic, Slobodan Obradovic, and Ljiljana Trajkovic. Performance evaluation of BGP anomaly classifiers. In *Digital Information, Networking, and Wireless Communications (DINWC), 2015 Third International Conference on*, pages 115–120. IEEE, 2015.

[144] Giuseppe Di Battista, Massimo Rimondini, and Giorgio Sadolfo. Monitoring the status of MPLS VPN and VPLS based on BGP signaling information. In *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pages 237–244. IEEE, 2012.

[145] Dominik Schatzmann, Bernhard Plattner, and Wolfgang Mühlbauer. Identification of Connectivity Issues in Large Networks using Data Plane Information.

[146] Cheng Qi Sun and Peng Fule Ding. Optimization Techniques of Traceroute Measurement Based on BGP Routing Table. In *Applied Mechanics and Materials*, volume 303, pages 2062–2067. Trans Tech Publ, 2013.

[147] Enis Karaarslan, Andres Garcia Perez, and Christos Siaterlis. Recreating a Large-Scale BGP Incident in a Realistic Environment. In *Information Sciences and Systems 2013*, pages 349–357. Springer, 2013.

[148] Philipp Richter. Classification of origin AS behavior based on BGP update streams. Master's thesis, Technische Universitat Berlin, 2010. Bachelor Thesis.

[149] Sara Anisseh. Internet Topology Characterizationon on AS Level. Master's thesis, KTH, School of Electrical Engineering, 10 2012.

[150] RIPE NCC. libBGPdump. https://bitbucket.org/ripencc/bgpdump, 2015.

[151] Dyn Research. Routing alarms. http://research.dyn.com/products/routing-alarms/.

[152] Mohit Lad, Dan Massey, Dan Pei, Yiguo Wu, Beichuan Zhang, and Lixia Zhang. Phas: A prefix hijack alert system. In *Proceedings of the 15th Conference on USENIX Security Symposium*, 2006.

[153] He Yan, Ricardo Oliveira, Kevin Burnett, Dave Matthews, Lixia Zhang, and Dan Massey. BGPmon: A real-time, scalable, extensible monitoring system. In *CATCH'09. Cybersecurity Applications & Technology*, pages 212–223. IEEE, 2009.

[154] Colorado State University. BGPmon. http://www.bgpmon.io/, 2015.

[155] Alberto Dainotti, Alistair King, Chiara Orsini, and Vasco Asturiano. BGP-Stream: a framework for BGP data analysis. https://ripe70.ripe.net/presentations/55-bgpstream.pdf, 2015.

[156] k. claffy. The 8th Workshop on Active Internet Measurements (AIMS8) Report. *ACM SIGCOMM Computer Communication Review (CCR)*, Jul 2016.

[157] CAIDA. BGPStream. https://github.com/CAIDA/bgpstream, 2016.

[158] Jim Cowie. The New Threat: Targeted Internet Traffic Misdirection. http://research.dyn.com/2013/11/mitm-internet-hijacking/, 2013.

[159] Alberto Dainotti. HIJACKS: Detecting and Characterizing Internet Traffic Interception based on BGP Hijacking. http://www.caida.org/funding/hijacks/, 2014. Funding source: NSF CNS-1423659.

[160] CAIDA. CAIDA BGP Hackathon 2016. https://www.caida.org/workshops/bgp-hackathon/1602/, 2016.

[161] CAIDA. BGPStream. https://bgpstream.caida.org/, 2016.

[162] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[163] Lixin Gao and Feng Wang. The extent of as path inflation by routing policies. In *Global Telecommunications Conference, 2002. GLOBECOM'02. IEEE*, volume 3, pages 2180–2184. IEEE, 2002.

[164] Craig Labovitz, Abha Ahuja, Srinivasan Venkatachary, and Roger Wattenhofer. The Impact of Internet Policy and Topology on Delayed Routing Convergence. In *20th Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, April 2001.

[165] NetworkX Developers. NetworkX. https://networkx.github.io, 2015.

[166] RIPE NCC. RIPE Atlas: A Global Internet Measurement Network. *The Internet Protocol Journal*, 18(3), September 2015.

[167] W. Kumari and D. McPherson. Remote Triggered Black Hole Filtering with Unicast Reverse Path Forwarding (uRPF). RFC 5635 (Informational), Aug 2009.

[168] R. Chandra, P. Traina, and T. Li. BGP Communities Attribute. RFC 1997 (Proposed Standard), Aug 1996. Updated by RFC 7606.

[169] Richard Steenbergen and Tom Scholl. BGP Communities: A Guide for Service Provider Networks . NANOG 40, Bellevue, Washington, June 2007.

[170] Benoit Donnet and Olivier Bonaventure. On BGP communities. *SIGCOMM Comput. Commun. Rev.*, 38(2):55–59, 2008.

[171] Christoph Dietzel, Anja Feldmann, and Thomas King. Blackholing at ixps: On the effectiveness of ddos mitigation in the wild. In *Passive and Active Network Measurement (PAM)*, pages 319–332. Springer, 2016.

[172] ARBOR Networks. ATLAS Q2 2015 Global DDoS Attack Trends. https://resources.arbornetworks.com/h/i/110843942-atlas-q2-2015-global-ddos-attack-trends, 2014.

[173] Cisco Systems. Remotely Triggered Black Holed Filtering. http://www.cisco.com/c/dam/en_us/about/security/intelligence/blackhole.pdf, 2005.

[174] J. Durand, I. Pepelnjak, and G. Doering. BGP Operations and Security. RFC 7454 (Best Current Practice), Feb 2015.

[175] Emile Aben. Has the Routability of Longer-than-/24 Prefixes Changed? https://labs.ripe.net/Members/emileaben/has-the-routability-of-longer-than-24-prefixes-changed, September 2015.

[176] Randy Bush, Olaf Maennel, Matthew Roughan, and Steve Uhlig. Internet optometry: assessing the broken glasses in internet reachability. In *Proceedings of the 9th ACM SIGCOMM conference on Internet measurement conference*, pages 242–253. ACM, 2009.

[177] D. Turk. Configuring BGP to Block Denial-of-Service Attacks. RFC 3882 (Informational), Sep 2004.

[178] CAIDA. Supplemental data: BGPStream: a software framework for live and historical BGP data analysis. http://www.caida.org/publications/papers/2016/bgpstream/supplemental/, 2016.

[179] Xiaoliang Zhao, Dan Pei, Lan Wang, Dan Massey, Allison Mankin, S. Felix Wu, and Lixia Zhang. An analysis of bgp multiple origin as (moas) conflicts. In *Proceedings of the 1st ACM SIGCOMM Workshop on Internet Measurement*, IMW '01, pages 31–35, New York, NY, USA, 2001. ACM.

[180] Bruno Quoitin, Cristel Pelsser, Louis Swinnen, Olivier Bonaventure, and Steve Uhlig. Interdomain traffic engineering with bgp. *Communications Magazine, IEEE*, 41(5):122–128, 2003.

[181] Doug Madory. The Vast World of Fraudulent Routing. http://research.dyn.com/2015/01/vast-world-of-fraudulent-routing/, 2015.

[182] Apache Kafka. http://kafka.apache.org/, 2015.

[183] Alberto Dainotti and Kc Claffy. Detection and analysis of large-scale Internet infrastructure outages (IODA). http://www.caida.org/funding/ioda/, 2012. Funding source: NSF CNS-1228994.

[184] Alberto Dainotti. North Korean Internet outages observed. http://blog.caida.org/best_available_data/2014/12/23/north-korean-internet-outages-observed/, 2014.

[185] Alberto Dainotti and Vasco Asturiano. Under the Telescope: Time Warner Cable Internet Outage. http://blog.caida.org/best_available_data/2014/08/29/under-the-telescope-time-warner-cable-internet-outage/, 2014.

[186] Sean Gallagher. Iraqi government shut down Internet to prevent exam cheating? http://arstechnica.com/tech-policy/2015/06/iraqi-government-shut-down-internet-to-prevent-exam-cheating/, 2015.

[187] Dyn Research. Iraq has had 12 govt-directed Internet blackouts since 27-Jun. https://twitter.com/DynResearch/status/629393185517666305, 2015.

[188] Doug Bernard. Iraqi Internet Experiencing ’Strange’ Outages. http://www.voanews.com/content/iraqi-internet-experiencing-strange-outages/2921135.html, 2015.

[189] Exa-Networks. ExaBGP. https://github.com/Exa-Networks/exabgp, 2015.

Appendix A

Papers and Prices

Conference Publications

- D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar,S. Sabyasachi, "*YouLighter: An Unsupervised Methodology to Unveil YouTube CDN Changes*", In: 27th International Teletraffic Congress (ITC 27), Ghent, Netherlands, 8-10 September 2015. **This paper is part of Cap. 2.**
- D. Giordano, S. Traverso, M. Mellia "*Exploring Browsing Habits of Internauts: a Measurement Perspective*", In: Asian Internet Engineering Conference 2015 (AINTEC 2015), Bangkok, Thailand, 18-20 November 2015. **This paper is presented in Cap. 3.**
- D. Giordano, D. Cicalese, A. Finamore, M. Mellia, M. Munafò, D. Rossi, D. Joumblatt "*A First Characterization of Anycast Traffic from Passive Traces*", In: Traffic Monitoring and Analysis Workshop 2016 (TMA 2016), Louvain La Neuve, Belgium, 5-6 April 2016. **This paper is presented in Cap. 4.**
- L. Vassio, H. Metwalley, D. Giordano, "*The Exploitation of Web Navigation Data: Ethical Issues and Alternative Scenarios*", In: XII Conference of the Italian Chapter of AIS, Rome, Italy, 9-10 October 2015
- C. Orsini, A. King, D. Giordano, V. Giotsas, A. Dainotti "*BGPStream: A Software Framework for Live and Historical BGP Data Analysis*", In: Internet Measurement Conference 2016 (IMC 2016), Santa Monica, USA, 14-16 November 2016. **This paper is presented in Cap. 6.**

Journal Publications

- D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar,S. Sabyasachi, "*YouLighter: A Cognitive Approach to Unveil YouTube CDN and Changes*", In: IEEE Transactions on Cognitive Communications and Networking doi: 10.1109/TCCN.2016.2517004. **This paper is presented in Cap. 2.**

- D. Apiletti, E. Baralis, T. Cerquitelli, P. Garza, D. Giordano, M. Mellia ; L. Venturini, "*SeLINA: a Self-Learning Insightful Network Analyzer*", In: IEEE Transactions on Network and Service Management doi: 10.1109/TNSM.2016.2597443. **This paper is presented in Cap. 5.**

Awards

- **Best student paper award:** D. Giordano, S. Traverso, L. Grimaudo, M. Mellia, E. Baralis, A. Tongaonkar, S. Sabyasachi, "*YouLighter: An Unsupervised Methodology to Unveil YouTube CDN Changes*" In: 27th International Teletraffic Congress (ITC 27), Ghent, Netherlands, 8-10 September 2015

- **Travel grant award:** At: 27th International Teletraffic Congress (ITC 27), Ghent, Netherlands, 8-10 September 2015

- **Travel grant award:** At: Traffic Monitoring and Analysis Workshop 2015 (TMA 2015), Barcellona, Spain, 23-24 April 2015

- **BGP Hackaton winning team:** R. Anwar, D. Cicalese, D. Giordano, B. Machado ,K. Nishizuka, N. Vivet, "*HIJACKS-2*, CAIDA BGP Hackathon 2016, San Diego, USA, 6-7 February 2016

- **Travel grant award:** At: Internet Measurement Conference 2016 (IMC 2016), Santa Monica, USA, 14-16 November 2016