TODO-FILL-HERE

EDITE - ED 130

THÈSE

pour obtenir le grade de docteur délivré par

TELECOM ParisTech

Spécialité "Informatique et réseaux"

présentée et soutenue publiquement par

Claudio IMBRENDA

le 28/11/2016

ANALYSING TRAFFIC CACHEABILITY IN THE ACCESS NETWORK AT LINE RATE

ANALISER LA CACHEABILITÉ DU TRAFIC DANS LE RESEAU D'ACCES AU DÉBIT DE LIGNE

Directeur de thèse: **M. Dario ROSSI, Professeur**, *Telecom ParisTech*
Co-encadrement de la thèse: **M. Luca MUSCARIELLO, Docteur**, *Cisco*

Jury

M. Marco MELLIA, Associate Professor, *Politecnico di Torino*	Rapporteur
M. James ROBERTS, Senior Researcher, *IRT System X*	Rapporteur
M. Daniel KOFMAN, Professeur, *Telecom ParisTech*	Examinateur
M. Fabio MARTIGNON, Professeur, *Université Paris-Sud*	Examinateur
M. Laurent VIENNOT, Directeur de Recherche, *Université Paris Diderot*	Examinateur
M. Philippe ROBERT, Directeur de Recherche, *INRIA*	Examinateur

THÈSE

TELECOM ParisTech
école de l'Institut Télécom - membre de ParisTech

Acknowledgements

I want to thank everybody who in any way, directly and indirectly, helped and supported me during this thesis.

A special thanks goes to all my ex-colleagues at Orange, for their support and friendship, and for the very warm and welcoming working environment they created. Another special thanks goes to my new colleagues at IBM, for their support and friendship, and for making it possible for me to continue working on this thesis.

A very special thanks goes to my family, for their support and help. They always supported and motivated me, even during the hard times... and always made sure there was some mortadella whenever I returned home to visit! A special thought goes to my grandmothers: the one who is very happy and proud of me, and the one who would be, hadn't she left us midway through.

Another very special thanks to my friend Thomas, for his help with technical problems, for the support and motivation, and for thoroughly proofreading this thesis multiple times.

Finally, the most special thanks goes to my very dear friend Silvia, for all the help she gave me even before I started the thesis, when I was searching for an apartment in Paris, and then for listening, for motivating me, for helping managing the stress, and finally for proofreading.

Abstract

Web traffic constitutes the largest part of the total traffic in the networks of Internet Service Providers (ISP), and caching surely looks like a promising way to both reduce the load on the ISP networks and to improve the user experience. In this thesis we assess the potential gains for ISPs of caching at the edges of the network, ideally by using an Information Centric Networking (ICN) approach.

In order to quantify the aforementioned gains, we needed to perform measurements of real traffic in the locations where we propose to place caches. We had the unique opportunity to place our own probe in the live network of Orange in Paris, in two different locations in the access network. There, using a high-speed and high-performance capture and Deep Packet Inspection (DPI) analysis tool developed from scratch during this thesis, we captured and analysed the live Web traffic of several thousands of Orange customers.

We needed to further analyse the results from the probe to assess the potential efficacy of caching and to determine their optimal sizes and locations in various levels of the access network, from the user to the DSLAM. To do so, we measured the amount of requests and traffic that could potentially be cached in the access network by using the widespread metrics of Cacheability and Traffic Reduction. We found the best timescale for the cacheability analysis and we estimated the appropriate cache size needed, which was then used to perform simulations with a cache simulator.

Hadoop was used to process the enormous amount of data produced by the tool, since traditional non-parallel and non-distributed strategies proved to be far from optimal. Moreover, in order to be able to process future datasets with a longer duration or a broader number of clients, it is necessary to be able to easily scale horizontally, which is not possible at all with a traditional analysis tool. We analysed the performance and accuracy of different object identification strategies used to generate the statistics. We also compared the Hadoop implementation with PIG and a classic sequential implementation.

In the first part of this thesis we introduce the innovative analysis tool and the statistics used; in the second part we find the right timescale for the statistics and we quantify the errors due to assuming a stationary catalogue and a Zipf popularity distribution; in the third part we use Hadoop to calculate the statistics, we explore the tradeoff between simplicity and efficiency of different Hadoop based systems, and we compared the advantages and disadvantages between Hadoop and a non-parallel, non-distributed analysis tool. We conclude that caching is possible even in the fast-path and at the edge of the network, thus showing that caching in the access network is a real opportunity for ISPs.

Keywords: Caching – Content Delivery Networks – Information Centric Networking – Traffic Analysis – Cacheability – Big Data

Résumé

Dans les réseaux des ISPs (Internet Service Providers – fournisseurs d'accès à internet) le trafic web est la plusparte du trafic total, et cacher semble sûrement un moyen prometteur pour reduire le charge sur les reseaux des fournisseurs et pour ammeliorer l'experience utilisateur. Dans cette thèse nous souhaitons évaluer les avantages potentiels pour les ISP pour cacher dans les bord extern du réseau, idéalement en utilisant un approche ICN (Information Centric Networking – Reseau Centré sur l'Information).

Pour quantifier ces avantages, nous avons besoin d'effectuer des mesures de trafic réel dans les locations où nous proposons de cacher. Nous avons eu la possibilité unique de placer notre sonde dans deux locations differentes dans le reseau d'accès de Orange à Paris. Là, en utilisant un outil d'analyse à hautes prestations qui effectue DPI (Deep Packet Inspection – Inspection Aprofondie des Paquets), dévéloppé entièrement pendand cette thèse, nous avons colleté et analysé le trafic Web de diverses milliers de clients Orange.

Nous avions besoin d'analyser ulteriorment les resulats de la sonde pour déterminer l'efficacité potentielle de cacher et pour déterminer la taille de la cache optimale et les locations dans le reseau d'access, du client au DSLAM. Pour ça, nous avons mesuré la quantité de requêtes et de trafic qui peut être caché par une cache placé dans le reseau d'accès en utilisant les métriques de Cacheabilité (Cacheability) et Reduction de Trafic (Traffic Reduction). Nous avons alors estimé l'échelle temporelle meilleure pour l'analyse de la cacheabilité et nous avons estimé la taille necessaire pour la cache, utilisée en suite pour effectuer une simulation avec un simulateur de cache.

Nous avons utilisé Hadoop pour processer les quantitées enormes de données générées par l'outil d'analyse, vu que des strategies non-paralleles et non-distribuées n'etaient pas optimales. En outre, pour pouvoir processer des dataset futurs plus longs ou avec plus de clients, il est nécessaire de pouvoir facilement scaler ?? horizontalement, et ça n'est pas possible avec toutes outils d'analyse traditionels. Nous avons analysé les prestations et la precision des differentes strategies d'identification des objets utilisées pour generer les statistiques. Nous avons aussi comparé l'implementation Hadoop avec PIG et avec une implementation classique sequentielle.

Dans la première partie de cette thèse, nous introduisons l'outil d'analyse innovant et les statistiques utilisées ; dans la deuxième partie nous trovons l'échelle temporelle correcte pour les statistiques et nous quantifions les erreurs dus à l'assumption d'un modèle avec catalogue fixe et distribution de popularité Zipf ; dans le troisième nous utilison Hadoop pour calculer les statistiques, nous explorons le compromis entre simplicité et efficacité dans des systèmes basées sur Hadoop, et nous avons comparés les avantages et disavantages entre Hadoop et un outil non-parallele et non-distribué. Nous concluons que cacher est possible, même dans le parcours vite et dans le bord du reseau, donc montrant que cacher dans le reseaux d'access est une opportunité réelle pour les ISPs.

Mots-clés : Cacher – Reseaux de Livraison des Contenues – Reseau Centré sur l'Information – Analyse du Trafic – Cacheabilité – Mégadonnées

Contents

List of Figures

List of Tables

Chapter 1

Introduction and motivation

Internet traffic is nowadays skyrocketing, and more and more services are being served through the web using HTTP[1]. A growing number of people are using more and more connected devices, which, due to technological progress, have an increasing amount of bandwidth at their disposal. The amount of 2G mobile connections is decreasing in favour of faster 3G and 4G, and in a few years ultra fast 5G mobile internet will be available for consumers.

This huge increase in available bandwidth spurred the creation and/or growth of bandwidth intensive applications, like Video on Demand (VoD) and Subscription Video on Demand (SVoD), such as Netflix or Youtube. Naturally many other applications also appeared, like file sharing or Voice over IP (VoIP), but they only constitute a marginal share of the total traffic, either because of the nature of the application, or because of throttling. Moreover they are not normally served over HTTP, so they are not relevant to this thesis.

Video streaming thus constitutes a huge part of internet traffic, and while in the past such services relied on proprietary transport protocols (like RTP and RTSP), streaming is nowadays performed almost exclusively over HTTP. By 2019, Cisco estimates that 80% of internet traffic will be due to video streaming[2].

The spread of Web-based content delivery solutions has multiple root causes (e.g. enhanced flexibility and ease of access from a vast set of user terminals, mostly mobile) and consequences. In particular, today Subscription Video on Demand (SVoD) consumers expect high Quality of Experience (QoE) when accessing their videos from any device inside and outside their home (TV, HDTV, smart-phones, tablets, media players).

1.1 Content Delivery Networks (CDNs)

At the end of the 20th century, Internet backbones were not as fast as they are today and clients had significantly slower connections (dialup was still common). Websites, especially those geographically far from clients were slow and had huge latencies. Content Delivery Networks (CDNs) were born to address this issue. The idea is straightforward: disseminate caches around the world in order to assure that a cache is always close enough to the clients; the clients are directed to a nearby cache depending on their location. CDN operators get revenue from the owners of the sites that use their network. This idea, initially developed only to increase the performance of end-customer-facing websites, also has the benefit of reducing traffic in the backbones.

Nowadays, fast intercontinental links allow for almost instantaneous access to any website anywhere in the world, thus diminishing the impact of their original task. But in the meantime CDNs have evolved to provide additional services, thus increasing their relevance once again.

First of all, CDNs provide high availability and high performance even in case of massive amounts of traffic, since the origin servers (the ones hosting the original content) will only see a very tiny fraction of the requests. This allows the publishers to directly manage small (and therefore cheap) servers. Having many caches distributed globally also helps against Distributed Denial of Service (DDoS) attacks, since multiple clients spread out geographically trying to hit the same server will in fact hit different caches.

The biggest CDNs operators are now putting their caches directly inside the networks of Internet Service Providers (ISPs), in order to further minimise the latency, at the cost of paying the ISPs for hosting the caches in their networks.

The caching advantages of CDNs also imply that websites will generate less traffic on the backbone, yielding savings for the website operators, since they have to pay for less traffic on the backbone, for the ISPs, since their backbone links now do not need as many upgrades, and finally for the CDN operators, because they get paid for their services.

Still, even when placed directly in the network of ISPs, CDN caches are usually very large and serve hundreds of thousands, or even millions of clients.

Since CDNs allow ISPs to save traffic, the ISPs themselves started to deploy their own CDNs to cash on the advantages; ISP-run CDNs are typically used for VoD, SVoD and TV broadcast. Traditional IP multicast fails to satisfy the large set of requirements needed: multicast is not reliable because it's UDP only, it does not have any form of congestion control, and finally and very importantly, multicast is also often misconfigured, and one single misconfigured node along the path is enough to render it useless. This forces ISPs to build their own video services on top of CDN systems, and by running the CDNs themselves, the ISPs can better optimise the content for their clients and networks.

Section 2.1 contains references to related works about CDNs.

1.2 Information Centric Networking (ICN)

Since web traffic is the overwhelmingly biggest share of the total internet traffic, the classical point-to-point paradigm of the net turns out to be sub-optimal. A better approach, taking inspiration from the ideas that drive CDNs, is Information Centric Networking (ICN).

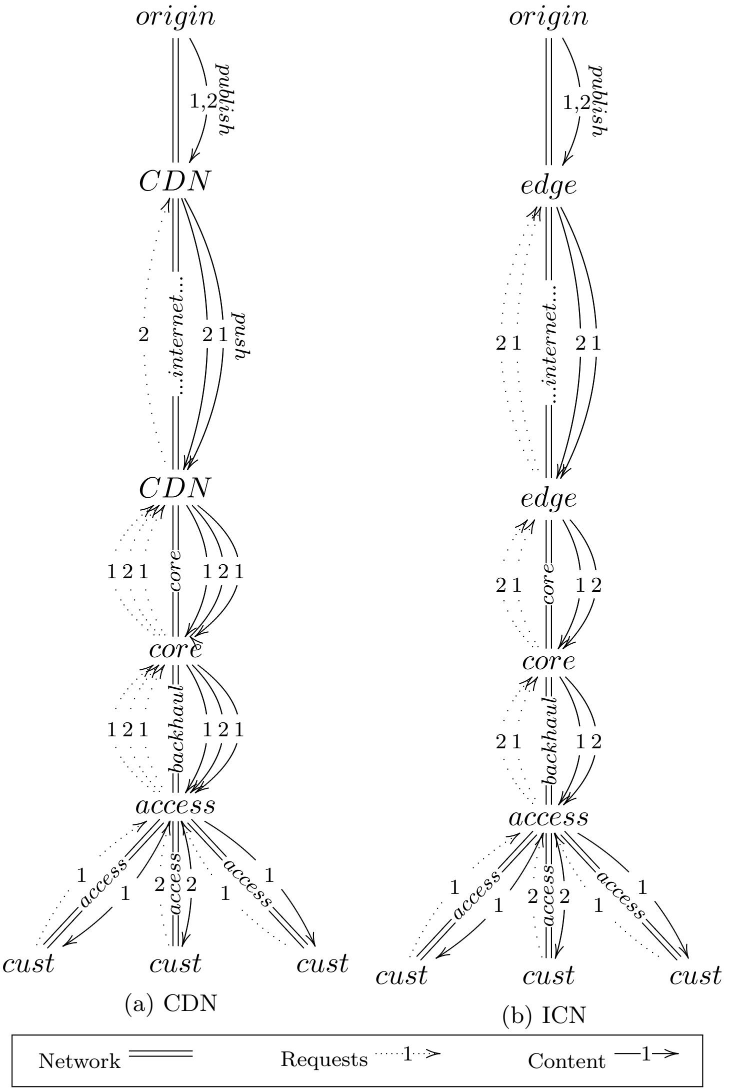
The fundamental idea of ICN is that the content is the basic unit; packets on the wires are either content or requests for specific content. There are no connections between a client and a server, just requests sent to the network, and replies with the requested objects. Each ICN router has a small cache (usually called Content Store); this allows to cache popular objects in the network locations where they are most needed. This also allows for interesting mesh-like and/or delay tolerant networks.

Having a global network of caches is almost like having a huge global CDN that serves all websites; popular content will be close to the end-users with high probability, origin servers are spared most of the load (thus handling high traffic or DDoS automatically out-of-the-box). The differences with actual CDNs are

- less control: ICN does not give control to the originator of the content about where the content will be cached
- size: most caches will be small, especially near the edges of the network; this might not be efficient
- no more CDNs: if every object is automatically cached in the network by the network itself, there is no more need for CDNs, at least in the form we know them currently.

This is indeed a huge paradigm shift, like the one from circuit networks to packet switching, and requires huge changes in the software and in the internet infrastructure. The advantages and disadvantages of this paradigm need to be studied thoroughly, since only very good results can possibly lead to its adoption. In particular the effectiveness of caches with a small user fanout needs to be assessed.

Section 2.1 contains references to related works about ICN; figure 1.1 shows a comparison between the ICN and CDN object delivery models.



(a) CDN

(b) ICN

Figure 1.1: Object delivery model: CDN vs ICN. Example of the delivery of a very popular object (1) and a less popular one (2).

1.3 ISP dilemma

Today's model presents various business weaknesses since it relies on third-parties (e.g. Netflix) to provide web services, whose quality depends on someone else's network infrastructure (one or multiple ISPs). For video-centric services, the relation between investments and revenues tends to be unbalanced when third-party content providers do not recognise any additional revenue to the ISPs which, instead, shoulder investment and operational costs to deliver additional traffic shares at no incremental revenue. The end customer is not eager to pay more to solve the dispute among the two parties if this requires to lose Internet access flat rate. The dispute involving Netflix and Comcast, Verizon and AT&T in the USA is one good example displaying the difficulties and frictions in this market [3].

ISPs are then faced with a dilemma: increase the capacity towards the content providers, shouldering the costs, or providing a sub-par service to their customers, causing complaints and potentially lawsuits.

To solve the issues between ISPs and content providers, we propose a novel approach for ISP content delivery, combining the service flexibility of typical Web based CDNs with efficient redundant traffic reduction as in an IP multicast tree. A few technologies suitable to achieve this goal already exist, i.e. JetStream [4], AltoBridge [5] and also Open Connect [6]. However, they provide no standardised architecture, they do not interoperate and they do not support every kind of Web content. We believe that an ICN approach can satisfy the aforementioned goals, in particular we consider the content-centric networking (CCN) architecture (a.k.a. NDN)[7]; by embedding a service-agnostic content caching into the network architecture, CCN may provide a common underlying network substrate for the deployment of next generation CDN systems.

By means of measurement and statistical analysis, this thesis shows that caching in the access network, although not necessarily using ICN, is a real opportunity for ISPs.

1.4 Performing Cacheability Analysis

Deciding the location and size of the caches in a network is fundamental for maximising the efficiency and the gains. On the other hand, such decisions can only be performed with enough information about the requests that are issued on the network segment that is to be serviced by the cache. Gathering such information is definitely not trivial due to the huge quantities of data involved in the process to perform this very estimation. Compromise solutions are detrimental: for instance, *sampling* the requests process may lead to optimistically filter out a bulk of unpopular content, overestimating caching gains; *capping* the maximum data rate limits the analysis to the edge of the network where the lack of statistically significant multiplexing can adversely impact the estimation, leading to caching gains underestimation; operating on *flow records,* implies losing important details (e.g., range requests, cookies, etc.) for object identification, which could again lead to overestimation. Clearly, combining any of the above simplification can either lead to uncontrolled error amplification or compensation, which a scientifically sound study should take care of avoiding.

Fortunately, the last decades have seen a flourishing ecosystem of software tools [8, 9, 10, 11, 12, 13] that allow to naturally cope with huge data quantities and that are labelled with the name of “big data” solutions. The best known example is represented by the Map-Reduce paradigm, initially proposed by Google[14], of which the most popular implementation is Yahoo’s Hadoop[9]. Initially proposed for distributed log processing, the paradigm has been quickly applied to other areas, including network and traffic analysis [15, 16, 17, 18, 19, 20, 21]. The reasons of this success are clear, considering that, provided that the problem at hand is amenable to parallel computation, the Map-Reduce paradigm offers horizontal scalability. In other words, once a distributed analysis system has been developed, it can be applied to larger datasets by parallelising computation over enough hardware resources (i.e. CPU and memory). In the age of digital data deluge, and by reason of the expected increase of traffic rate, this property practically becomes a requirement to maintain computational feasibility (i.e. gathering results, in a furthermore useful time).

It follows that the problem of traffic cacheability estimation is worth attacking with a big data approach, which is precisely the aim of this work. Our previous work in this area has tackled the analysis of traffic cacheability with more traditional techniques. Yet we faced significant scalability challenges to apply the same methodology on larger dataset – where larger is intended here in temporal, spatial and line-rate respects. In this thesis we solve these scalability issues with Map-Reduce.

1.5 Contributions

We observe that caching is possible even in the fast-path of the network, thanks to the relatively small sizes needed for the caches. We reach this conclusion by analysing the live traffic of several thousands of Orange customers in Paris, using an analysis tool developed for the purpose.

The main contributions of this thesis are the following:

- Traffic analysis tool: design and development of a real-time analysis tool for high performance line speed dissection and analysis. Notably the tool can analyse HTTP transactions at 10Gbps with a single core. (Sec. 3.2).
- Largest dataset: we conduct our analysis on datasets one order of magnitude larger than the ones used in existing literature (Sec.3.3).
- Timescale analysis: inferring the timescale at which it makes sense to cache in the access network (Sec. 4.2.1). We find out that one day is the optimal timescale

for cacheability analysis at the access network.

- Content popularity analysis: analysing and characterising content popularity over a meaningful timescale (Sec. 4.2.2).
- Sizing of caches: determining a good value for a real-world cache in the access network, and validate it through simulations (Sec. 4.3).
- Object identification strategies: analysing the impact of different object identification strategies on the cacheability statistics. (Sec. 4.4).
- Map-Reduce framework: design and implementation of a scalable analysis system using the Hadoop Map-Reduce framework. (Sec. 5.2).
- Map-Reduce benchmarking: measuring the impact of different optimisations on the performance of Hadoop, comparison with PIG and classical sequential analysis (Sec. 5.3).

1.6 Organisation

This thesis is organised as follows. In Chapter 2 we present other works related to the topics of this thesis: about ICN and CDNs, about other internet traffic capture and analysis tools, about other cacheability analysis and about MapReduce frameworks. Chapter 3 introduces the capture tool developed during this PhD, the network topologies where it was deployed and the hardware it runs on, and the dataset it collected. Chapter 4 presents the cacheability analysis of real traffic, including timescale analysis and simulations of different scenarios. Chapter 5 focuses on the big-data approach needed to process the sheer amount of data collected, including parallelisation of cacheability analysis and optimisation for speed. Finally in Chapter 6 a perspective of the achieved results and future topics is given.

Figure 1.2 shows a workflow scheme of the capture and analysis process, referencing the chapters where the different parts are discussed. Notice that the presentation infrastructure was developed by Wuyang Li, and not by the author, its description is included in Appendix C for reference purposes.

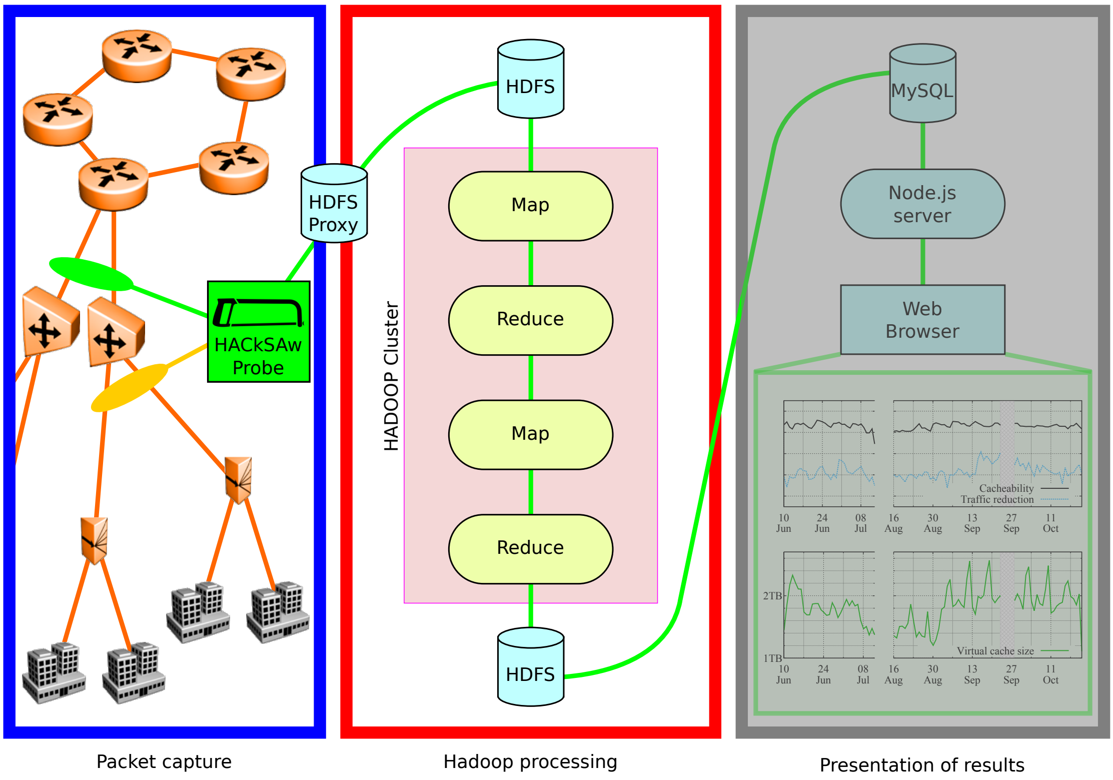


Figure 1.2: Workflow overview of the capture and analysis process. Chapters 3 and 4 introduce and describe the part on the left, Chapter 5 describes the middle part, while the presentation of the results is described in Appendix C.

Chapter 2

Related works

This chapter gives an overview on works related to the topics touched in this thesis. We start from ICN and CDNs, since there are many studies and articles about architectural simulations and cache placement, dimensioning, and decision. We will provide an overview on the existing traffic capture tools and explain why we needed to write a new tool. We will also present a quick comparison of the types and sizes of datasets collected in other works. Subsequently we will introduce some works that also perform cacheability analysis, although with different methodologies than us. Finally we will provide an overview of different existing Big Data and MapReduce frameworks.

2.1 ICN/CDN

In the last few years, a significant body of work has tried to measure the relation between the amount of resources and efforts to achieve the gains promised by CCN[22, 23]. Most of these works are based on network simulations in some very specific scenarios, none of them taking into account the ISP infrastructure. They focused instead on content placement and caching performance no deeper than the PoP (Point of Presence). Other works[24, 25], have shown significant gain of CCN in some relatively downsized network settings. The drawbacks of computer simulation and network experimentation are that, while being valuable, they do not allow to generalise the results for realistic workloads, which are difficult to accurately model and even synthesise. Analytical models of such systems [26, 27] allow to quickly evaluate the relation between QoS, network capacity and users' demand, but they either fail to provide reliable performance predictions or must be tuned, a posteriori, using a data sample [28],[29]. Additionally, precise values of Web content *size* are often neglected in previous works, which are based on the analysis on HTTP requests only, neglecting all HTTP replies and resulting in possibly biased evaluations; the most notable exceptions are [30] and [31], which use a measurement methodology similar to ours and analyse both HTTP request and replies in radio mobile networks. See chapter 3 for more details about the capture tool and the gathered datasets.

The problem of cache placement (i.e. where to cache), dimensioning (i.e. how much to cache) and decision (i.e. what to cache) have thus received much attention in recent literature. In [23] it is shown, by using trace driven simulations, that a simpler caching architectures also deliver sizeable advantages. [32] provides an analytical characterisation of bandwidth and storage caching under some generalistic assumptions regarding content demand, network topology, and content popularity. [33] gives a closed form formula to estimate the throughput as a function of parameters like hit ratio, content popularity, content and cache size, which is then validated with a chunk-level simulation. [34] provides a mathematical explanation of the Che approximation, showing that it holds for ICN. In [35] existing LRU approximation algorithms are used as basis for a new algorithm that can be used to approximate LRU cache networks; per-cache and per-network performance measurements and error analysis are also performed. [36] uses a YouTube-like catalogue to perform a performance evaluation of CCN, finding that a simpler caching architecture is good enough, and that the catalogue and the content popularity are fundamental parameters. By performing a packet-level simulation, [37] investigates the tradeoffs in ICN between forwarding requests to known copies of the object and towards unknown paths. Similarly, [38] investigates a dynamic request forwarding approach for ICN, similar to Q-routing. [39] demonstrates the advantages of stateful routing, in particular with respect to name prefix hijacking, failed link avoidance, and multipath routing. In [40] ICN simulations are performed with a YouTube-like catalogue and large caches at the nodes to asses the usefulness of topological information when allocating cache sizes. In [41] a probabilistic caching strategy for ICN is developed and compared to other caching strategies. In [42] a content caching scheme is proposed that allocates space in the caches in function of the popularity of the objects. On the other hand, [43] shows that random caching in a single node on the delivery path provides similar or even better results than pervasive caching, both on synthetic and real topologies. In [79] we show the advantages of caching at the edge of the network, like ICN would do natively, and we show that there are clear advantages. [44] shows instead that pervasive caching yields clear advantages when caching decisions and forwarding are tightly coupled, in particular when iNNR (ideal Nearest Replica Routing) and LCD (Leave a Copy Down) are jointly used. Finally [45] proposes a cost-aware caching strategy for

ICN, comparing it with non-cost aware strategies by means of numerical simulations.

In this context, network traffic measurement plays an important role in assessing the gains that caching technologies could bring to an Internet Service Provider (ISP), by e.g., estimating the attainable traffic reduction. A campus network is analysed in [46], in particular YouTube traffic, and the results of the measurements are then used to create a traffic generator. The traffic of over 20 000 residential DSL users is analysed in [47], finding that Peer-to-Peer traffic is more cacheable than HTTP. [48] instead focuses on estimating the common measurement errors that can be incurred when analysing HTTP traffic. [49] uses a real trace to show that pre-staging popular content directly on mobile phones could yield interesting bandwidth savings. Finally the already mentioned [31, 30, 29][79] are also focused on traffic measurement and analysis.

2.2 Capture tools

Recent works on web caching within the radio mobile backhaul in New York metropolitan area [30] and in South Korea [31] have used proprietary tools satisfying the mentioned requirements. Unfortunately, none of these tools is publicly available; the first one is an internal capture tool from the operator, and the second, while developed by the authors of that paper, was not released publicly. Conversely, popular open source tools like *bro* [50] and *Tstat* [51] do not satisfy all the necessary requirements for our setup. Indeed *bro*, conceived as an intrusion detection system, is not suitable for high speed monitoring because it applies regular expressions on each packet to detect signatures of known threats, and therefore results to be very slow. *Tstat*, instead, is faster and accurate in analysing TCP connections, but inaccurate in analysing HTTP transactions. Consequently, both tools turn out to be not satisfactory for our needs.

For this reason, the analysis presented in this thesis is based on a novel tool, called HACkSAw, that we developed to accurately and continuously monitor web traffic in a modern operational ISP network, at any line rate, for any workload. Unfortunately we couldn't release it publicly, as per company policy.

A comparison of the performance of the different tools, using the same one hour PCAP packet trace as benchmark, is reported in Tab.3.1, in chapter 3.

2.3 Datasets

Datasets of works inherent to caching or workload characterisation can be request logs from servers, like [52, 53] which analyse the traffic logs from VoD services, or [23] as already described previously. They can otherwise be gathered via active crawling techniques of popular portals, [54, 55] in particular analyse YouTube and compare its traffic to web or VoD traffic. Finally, datasets can be collected via passive measurement methodology [30, 31] as we do in this work.

A comparison of some basic information about the dataset considered in this and related work is given in Tab.3.2, in Chapter 3, where it can be seen that our datasets are not only the *longest in time* but also the *largest in volume* and the *smallest in population*. We therefore expect to gather statistically relevant results that additionally allow us to observe phenomena on timescale that were not observed in other studies focusing on shorter timescales and especially to get conservative estimate of caching gain by reason of the limited aggregation due to the small population size. Notice that Tab.3.2 additionally reports, for completeness, cacheability information collected in these studies; yet, the comparison is in this case only meant to be *qualitative*, as the dataset collection methodology, cacheability definition and analysis technique differ. It is however worth stressing that our cacheability results are in line with those gathered by [31], which is closest to this work in terms of methodology but radically different in terms of network environment.

2.4 Cacheability analysis

Since caching is a network primitive in most ICN systems (including CCN), most of the works on ICN perform some degree of cacheability analysis [22, 23, 24, 25][26, 27], although, as explained before, some perform simulations of very simple network topologies.

A common assumption to the evaluation of caching systems performance is to assume that content requests are generated under the IRM (Independent Reference Model[56]), with a request distribution following a Zipf law. Considerable measurement efforts have been devoted to provide input to this workload in static settings, while very few consider the catalogue dynamics over time. Characterisation of video catalogues has especially attracted significant attention, from YouTube crawling [54], to measurement at a campus [46] or ISP [28, 29], just to cite a few. Focusing on YouTube traffic, [28], [29] show that IRM assumption may yield to a significant under-estimation of the achievable caching gains. However, the main focus of [29] is to propose a fine grained characterisation of the data, rather than assessing the impact on the expected CCN gain (even a lower bound), as we do in this work.

Work closer to ours in terms of methodology is represented by [47, 48, 49, 31, 30, 23], which we described previously; in Sec. 5.1.2 we also provide a closer comparison between the datasets used in those work and the ones used in this thesis. At high level it is worth remarking that while other works have measured cache performance for video applications [57], [47], most of the previous work has neglected web content which has, however, a significant impact on the amount of required storage to install in the network. Additionally, precise values of Web content *size* are often neglected in previous works, which are based on the analysis on HTTP requests only, neglecting all HTTP replies and resulting in possibly biased evaluations; the most notable exceptions are [30] and [31], which use a measurement methodology similar to ours and analyse both HTTP request and replies in radio mobile networks.

2.5 MapReduce frameworks

This work focus on the analysis of traffic cacheability properties in operational networks. As such this work provides input to the large body of work that focuses on many aspects of CDN/ICN caching, such as CDN/ICN comparison [23, 44] modelling and performance evaluation [32, 33, 34, 35], object replica discovery[36, 37, 38, 39, 41, 42, 43].

However, our goals are orthogonal with respect to the design and evaluation of new CDN/ICN techniques, so that we deem the above work out of scope. Rather, two very far apart classes of work relate to this: on one hand, we have works whose focus is the monitoring and characterisation of traffic cacheability statistics[46, 47, 48, 49, 31, 30, 29][79]. On the other hand, we have works that employ big data frameworks[8, 9, 10, 11, 12, 13] for the purpose of traffic monitoring.

While our previous work [79] falls in the first category, this thesis is focused on the use of big data framework to scale up the analysis carried on in [79] for more limited datasets (i.e., in temporal, spatial and line-rate respects). See [79] for a more detailed comparison with the state of the art in this respect[46, 47, 48, 49, 31, 30, 29].

Work that is closer to the contribution of this thesis is work using big-data frameworks for the analysis of network traffic or properties. From a high level viewpoint, large-scale data processing techniques can be divided into two classes. On one hand, we have *stream processing* systems that operates over data nearly in real time: these includes both the general purpose systems (e.g. Storm, Spark) as well as systems specialised for the networking domain (e.g. Blockmon[8], DBStream[58]).

On the other hand, we have *batch processing* systems that operate over large datasets but are not suitable for real-time processing: a large family of these systems exists, which can be traced back to Google's MapReduce, that are surveyed in [59] and some of which (MapReduce[14], Stratosphere[10], Hama[11], Giraph[12], Graphlab[60]) are experimentally compared in [61].

Map-Reduce is by far the most popular system in the networking community, and Apache Hadoop is the most popular among several alternative implementations. The range of networking tasks Map-Reduce has been used for include initial log analysis[14] to analyse of large social graphs such as Wikipedia[21]. There are however fewer examples of work leveraging Hadoop MapReduce for the analysis of network traffic, which is thus closer to our work: [15] uses Hadoop to perform flow analysis, observing a very sizeable speedup compared to non-parallel implementations; in [16] Hadoop and PIG are used together with R to scale up the analysis of a distributed monitoring infrastructure; [17] performs IP, HTTP and NetFlow analysis of several TB of traffic traces; MapReduce is used in [18] to process traffic traces of a live UMTS network in China; [19] uses a Hadoop framework to process large traffic traces to detect anomalies in the network; finally [20] uses MapReduce to perform botnet detection in a peer-to-peer network, with data obtained from a distributed sniffing system.

Chapter 3

Capture tool and dataset

Cacheability analysis of internet traffic is obviously only feasible if it is possible to analyse actual traffic. The two obvious prerequisites are the access to an operational network and a probe to capture the traffic and extract in real-time the dataset to be processed later.

This chapter illustrates the network location where our probe is situated, the traffic analysis tool that runs on the probe, and the dataset it generates, which is then used to produce the results of the next chapters.

3.1 Network and capture points

We had the possibility to place our own hardware probe in a Central Office in the operational network of Orange in Paris. The GPON-based network is laid out as in picture 3.1. The GPON (Gigabit Passive Optical Network) tree passively aggregates up to 64 users on a single optical link, up to 16 such links are then aggregated at the OLT (Optical Line Terminal), with a backhaul link towards the core.

In 2014 we were downstream of two such OLTs each with a 1Gbps backhaul, as expected we observed less than 2 000 users. In 2015 we moved one step up in the network, we were upstream of two full-duplex 10Gbps links, and downstream of a BAS (Broadband remote Access Server). The links aggregate several OLT backhaul links, where we observed more than 30 000 users. In both cases the packets captured were still encapsulated in PPPoE, and presented VLAN tags.

Due to the type of analysis performed, we need to guarantee that the routing of the packets is symmetric, because we need to capture both the upstream and downstream of each connection. Both locations where the probe was installed fulfilled this requirement,

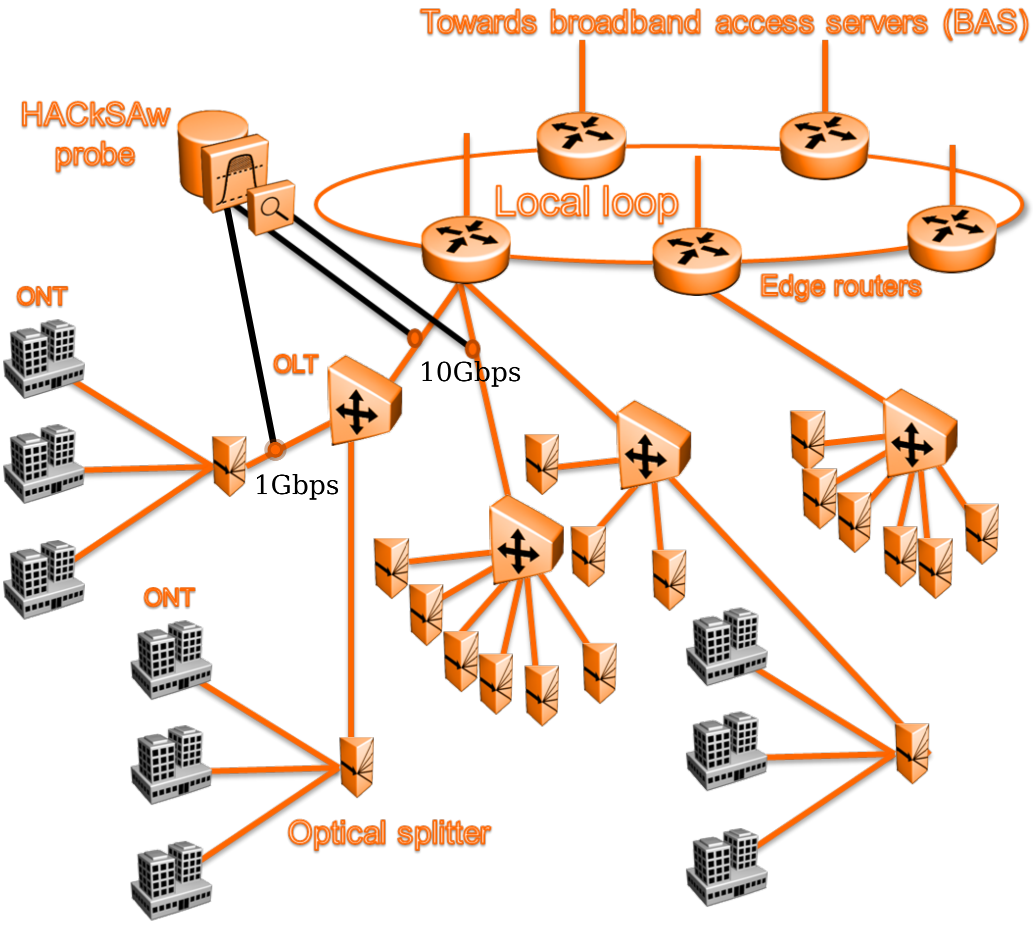


Figure 3.1: ISP network fibber access and back-haul. HACkSAw probes are deployed in different network positions (OLT vs ONT), corresponding to different link capacities (1Gbps vs 10Gbps) and levels of user aggregation (1,500 vs 40,000).

since they are the only links connecting the clients to the core of the network.

3.1.1 Hardware

The tool runs on an IBM server with two quad-core Intel Xeon E5-2643 CPUs at 3.30GHz with 48GB of RAM each, for a total of 96GB of RAM memory; thus it is a NUMA (Non Uniform Memory Access) setup. The server has 5 1-TB hard disks in a RAID-5 configuration, thus yielding 4TB of usable storage. Debian is the operating system installed.

The server is equipped with an Endace DAG 7.5G4 card (DAG 10X4-P for the 2015 datasets), capable of capturing packets from 4 links simultaneously, allowing us to monitor two full-duplex Gigabit (10Gigabit for the 2015 datasets) links.

The capture cards of the probe were connected to passive optical splitters. A relatively slow speed link allows us to interact with the probe remotely.

3.2 HACkSAw, the capture tool

This section introduces HACkSAw (HTTP And Connection Stream Analyser), the live traffic analysis tool developed during this thesis, and already also used in [79, 80, 81, 82, 83] to produce the traffic traces. In particular, the following points will be addressed:

— motivations for writing a new tool, when many traffic analysis tools are already available.

— evolution in time of the requirements.

— general architecture of the system and its evolution to satisfy the requirements.

— some of the implementation details.

— performance of the tool.

3.2.1 Motivations

Traffic analysis is surely not a new thing, and therefore many tools exist to perform the task. Some tools match our needs better than others. As mentioned previously (§2.2), some tools were already adequate to our requirements, but those tools are not available, and other available tools, on the other hand, didn't meet our needs, therefore it was necessary for us to write a new tool.

3.2.2 Requirements

The requirements of the tool changed quite dramatically during the course of this PhD. Initially we only had PCAP traces of traffic, so the initial version of the tool was an off-line analyser; no timing issues, no hardware interaction. Since it used the PCAP APIs to access the packets from the trace file, it was rather trivial to extend it to support live capture. Some architectural modifications were necessary in order to achieve good live analysis performance at 1Gbps, some more radical changes were then needed to scale up the analysis to 10Gbps. Additionally, DNS analysis was deemed necessary. So in the end the final requirements for the capture tool are:

— Online and offline analysis. The tool must still be usable on offline traces, but its main focus has to be live capture.

- PCAP and DAG APIs. DAG APIs are fundamental to achieve line speed at high speed, whereas PCAP are needed because they are very widespread, and many other capture libraries provide compatibility layers for PCAP. By supporting PCAP it is this possible to support all the existing and future systems that offer such a compatibility layer.
- Not excessively big memory footprint. The probe should not be required to be equipped with huge amounts of memory, for cost, efficiency, and scalability reasons.
- HTTP and DNS analysis. We need an in-depth analysis of HTTP and DNS transactions, including timing information, object sizes, and other headers and information.
- Easy to maintain and extend. Ideally the design should be modular, in order to allow for easy maintenance and extension.
- 10Gbps to 40Gbps (or even 100Gbps) line speed capture without packet loss. The tool should be able to analyse the traffic in different points of the network at line speed, including close to the core.
- Proper software release for internal use: configure script, makefile, startup scripts to run the tool as a daemon.

3.2.3 Architecture and fundamental design choices

HACkSAw is written in C for 64-bit Linux systems. C++ was considered and discarded because of the added complexity and overhead. Any other language is unfit for the task, since we are dealing with time critical tasks and interacting with C APIs anyway.

On speed grounds all parsing of all layers is done in plain C, therefore including the Ethernet layer (including VLAN and PPPoE), TCP/UDP/IP, DNS and HTTP.

The behaviour of the tool can be configured both with command line parameters and a configuration file. Appendix A contains a description of all the configuration options.

The output of the tool is a series of plain-text files, one for each protocol analysed. Each line represents a connection or a transaction, values are separated by spaces or by tabs. This is indeed not the most space efficient way to represent this information, but we wanted

- human readable output, in order to be able to easily look at the raw data to perform basic correctness checks and help during debugging
- plain-text output, in order to be able to manipulate the results using standard

UNIX command line utilities

— the minimum amount of complexity and overhead in the tool itself; compression is something that can be done externally (as we indeed do, see Chapter 5)

A complete description of the columns of each output file can be found in Appendix B.

The architecture of the tool evolved with time, the next two sections illustrate the first and the latest architectures that were actually used on live traffic to gather datasets and produce scientific results.

3.2.4 The initial version – 2×1Gbps

The first version of the tool was rather simple. Taking inspiration from [31], all TCP connections were reconstructed directly in main memory. Once the TCP connection was over, the reconstruction buffers were handed to the HTTP analyser. The PCAP API were used for capture, since they have a backend for the DAG cards used in our probe. Figure 3.2 shows a scheme of how the tool was structured.

Input

The PCAP APIs are not multithreaded, so a dispatching system was implemented to allow for multiple analyser threads in parallel. The producer thread receives each packet using the PCAP API, it analyses the packet superficially just enough to understand in which output bucket to copy the packet. To minimise the impact of locking between the producer thread and the worker threads, packets are grouped in batches of 1000 packets [1]. Once a batch is full, it is sent in a queue to the worker.

Processing

The workers themselves process the packets they receive from the batches, following and reconstructing the TCP session. After a few packets are received, the L7 protocol is detected. If the protocol is not supported, no further analysis is performed, the connection is marked so that no further reconstruction will happen, and the existing reconstruction buffers are freed. Once the TCP session has ended, or once it is inactive for too much time (by default 15 minutes, but it is configurable), the reconstructed buffers are handed to the HTTP analyser. The HTTP analyser, which runs in the same thread as the TCP analyser, analyses the full buffers, extracts all the relevant information regarding all the transactions and writes the relevant output in the log file.

[1] the value can actually be configured at compile time

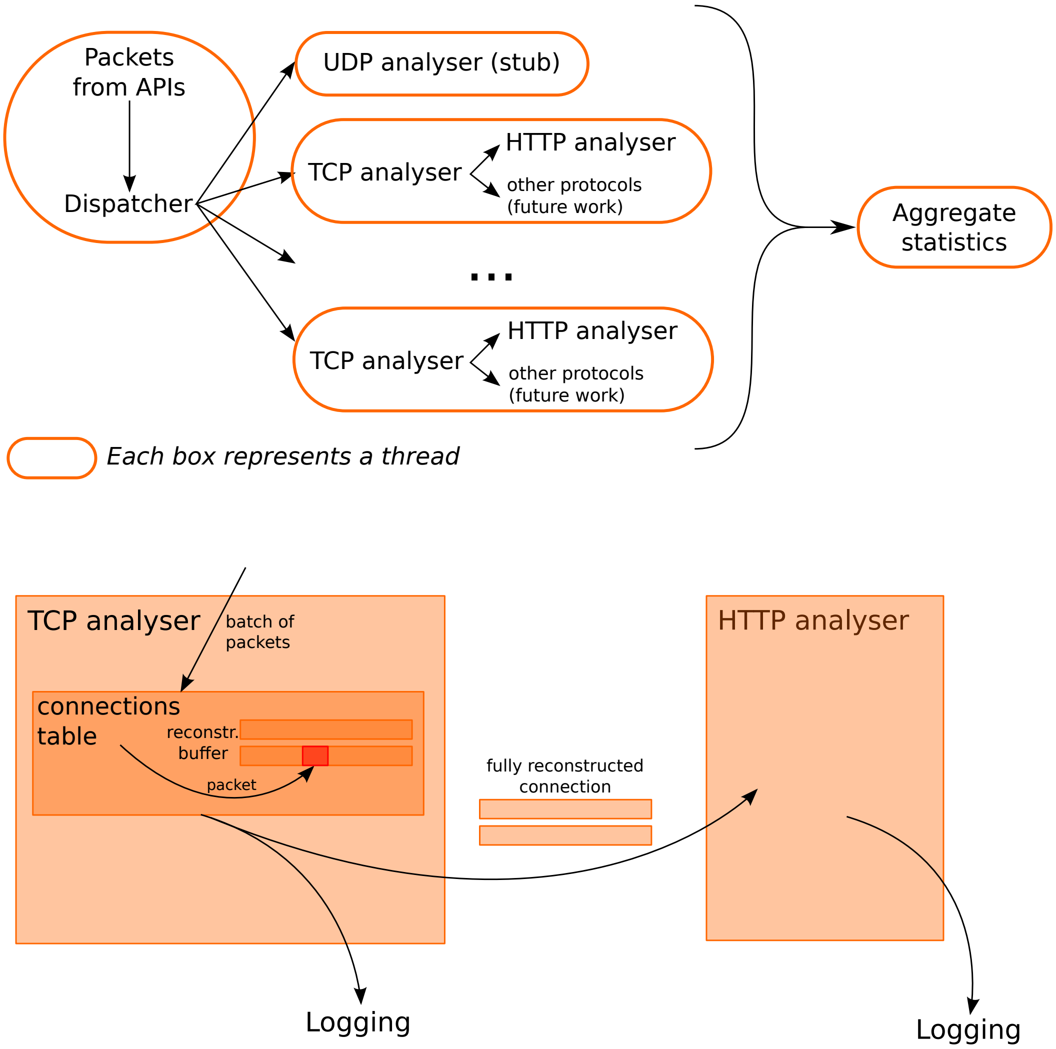


Figure 3.2: Diagram of operation of HACkSAw, first version.

The TCP threads need to periodically check for stale connections and terminate them. This causes periodic stalls in the threads which increase the latency, potentially stalling the producer thread. The queue of batches between the producer and the consumer threads exists to buffer those moments of higher CPU activity.

Output

Only two files were written: one for the TCP statistics, with one entry per TCP connection, and one for HTTP statistics, with one entry per HTTP transaction. Each entry is individually timestamped, but only two files are created and appended to as needed.

Shortcomings

This implementation presented many shortcomings. The most evident one is that the producer thread (dispatcher) does not scale. It has to process and copy *every* packet received from the capture API. The processing overhead is minimal, the real bottleneck is copying the packet from the receive buffer to the packet queues. A small stopgap measure involves accepting only TCP packets and dropping every other packet early. This speeds up significantly the process at the expense of a negligible increase in the processing time. This was still not enough to scale up more than 14 Gbps.

The second huge problem is the incredible amount of memory needed. Since every HTTP connection was completely reconstructed in memory before being analysed, even small capture points with few users and little traffic used up all the 96GB of RAM of the probe (and often spilling into swap, with catastrophic impact on performance)

A third shortcoming is that the output is not chunked in manageable chunks, and therefore the output files grow to unmanageable sizes. Performing most tasks on the output requires reading the whole file. When the file size in in the terabytes range, it is highly inefficient. Moreover it is not possible to selectively compress or transfer logs from previous days or hours.

3.2.5 The final version – 2×10Gbps and beyond

The final version addresses all the shortcomings of the first version, and furthermore adds some features to improve the maintainability and extensibility of the system. Figure 3.3 shows how the final version of the tool is structured.

Input

First of all, the DAG API is now used directly. This allows to use multiple hardware queues. The DAG card hashes the addresses and places the packet in the right queue, which is read by the consumer thread directly, thus removing the dispatcher from the picture. Without the bottleneck this design can now literally scale linearly with the number of CPUs. This solution was unfortunately not useful for us, as the address hashing does not work if the packets still have the PPPoE header. Another possibility is to dispatch the packets in the hardware queues based on the physical port on which they were received; this not only guarantees that both directions of any stream are assigned to the same thread, but additionally it also guarantees that correlated traffic is also

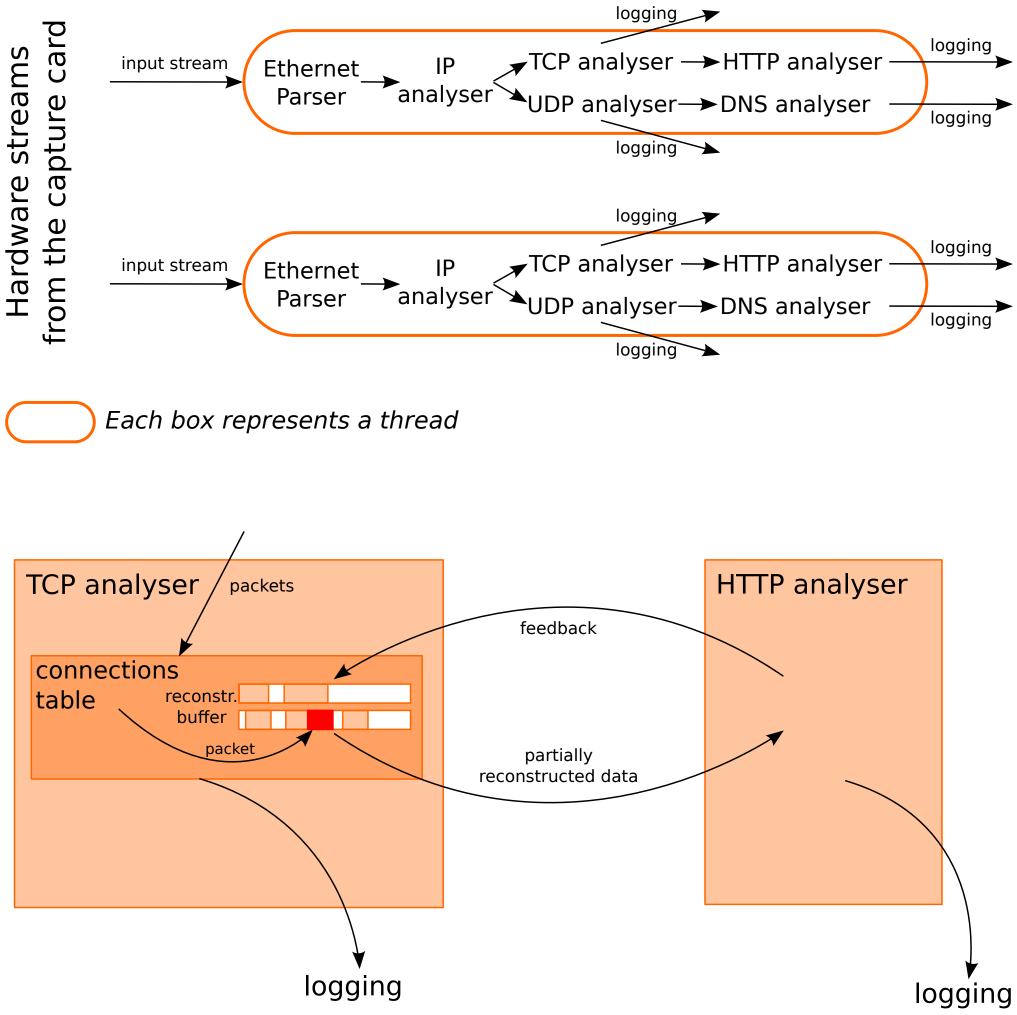


Figure 3.3: Diagram of operation of HACkSAw, final version.

assigned to the same thread. For example, using the hash of the IP addresses, it is possible that the DNS transaction used to resolve a name subsequently used to establish an HTTP connection could be assigned to a different thread than the HTTP connection itself; dispatching packets based on hardware queues solves this problem, and allows to embed some DNS information in the HTTP log, thus enormously easing postprocessing.

The input API is now modular, and the input module can be set in the configuration file or in the command line. There is one built-in module (`pcap_file`), and two more that are compiled if the relevant APIs are available (`dag` and `pcap`). The dispatcher bottleneck thread is still present when using the `pcap` input module. The `pcap_file` uses `mmap` to map the whole input file; the individual worker threads will then process all the packets, skipping over the ones that are not meant to be assigned to them.

There is now a clear internal API for input modules, so that adding new input module literally only requires to add the new implementation in a new file and adding the implementation in the list of available inputs.

Processing

All supported layers (L3, L4 and L7) are now modular, with an internal API to detect the protocol and instantiate the right analyser. This allows to easily implement and add new protocol analysers without having to perform changes spread out all over the place. Protocols can be enabled or disabled from the configuration file or from the command line.

To reduce memory consumption and to increase processing speed, a new internal API was implemented to allow communication between the L4 and L7 layers. This system allows to only keep the data that will be needed, and allows to skip anything that is not needed. The process can be simplified as follows:

1. Packets are collected by the L4 analyser (e.g. a TCP connection carrying an HTTP transaction)
2. The collected packets are handed to the L7 analyser
3. The L7 analyser can reply asking for more input (e.g. incomplete headers, return to point 1), or it can reply with a new target position in the stream (e.g. the headers were parsed, and we skip over the actual content of the object)
4. The L4 will discard all saved and incoming packets up to the new position; once a packet arrives at the new position, continue to point 1
5. The L4 will notify the L7 analyser when a packet was lost (if it wouldn't have been discarded anyway)
6. The L7 tries to handle the situation anyway, in case of success, return to point 3 (e.g. if the lost packet was not important, or maybe the L7 protocol analyser can cope with lost packets)
7. In case of failure, the L7 returns a failure code that causes the L4 to drop all saved packets and skip all subsequent incoming packets (thus never calling the L7 again)
8. The end of the stream is handled like a packet loss

All this complex interaction is implemented with only two functions per direction (so four in total). The semantic of those two functions can be summarised as:

- L7 should process this data and tell the new target offset to L4.
- L7 should process this data and tell the new target offset to L4, but if the new offset is smaller than the offset where the packet was lost, L4 will set the new target offset anyway, discarding potentially important data.

Output

The output is also modular. The default (and so far only) output module is the plaintext module, but other output modules (e.g. binary, database, hadoop) are easy to implement and add.

The plaintext module now splits each output file in chunks of configurable size. For example, it is possible to have the logs chunked by day or by hour. This allows to transfer, compress, and process specific time intervals, even while the software itself is still running.

The plaintext module also has an increased buffer size, thus reducing the number of system calls to be performed. Writing to disk thousands of times per second has a negative impact; decreasing the amount of system calls and filesystem operations by one or two orders of magnitude definitely has a positive impact on performance.

Advantages

First of all, removing the bottleneck allows to scale up linearly with the number of available CPUs. The complex stateful system of L4/L7 analysers allows to discard all unneeded packets, thus speeding up even more. The increased output buffer mitigates the impact of performing system calls, which was not a dominant factor, but it still contributes to improve the overall performance.

3.2.6 Some implementation details

Hash function for the dispatcher

All hash functions are applied to the XOR of the source and destination IP addresses. This guarantees that both directions of the stream are assigned to the same worker.

Several hash functions were tried for the dispatcher thread. The first implementation used a FNV-1a hash of the XOR of the source and destination IP address. Source and destination ports were not taken in consideration because they added undue complexity to the dispatcher thread. Since the dispatcher thread is the bottleneck of the architecture, it is important that the least amount of work be performed on the critical path.

Some tests showed that using just the XOR of the least significant bytes of the source and destination IP addresses was just as good, so that is the hash function used now when the PCAP APIs are used. Using more than just the last byte yields no additional benefits, and since we perform a division, whose remainder is then used to decide the thread for the packet, having a smaller integer will speed up the calculation.

Slab allocator

Memory is allocated and deallocated quite often in the analyser threads, and this puts a lot of pressure on the memory allocator. The standard allocator does a rather good job in general, but its performances can be improved upon, by implementing and using purpose-tailored allocators.

The logic behind the custom allocator is actually rather simple. Each thread has its own independent allocator. Each allocator keeps a list of free memory chunks (called *free list*) for a range of sizes that is used often in the program. When an allocation is performed, the first free item is removed from the right free list and returned; conversely when a memory area is freed, it is placed back on the head of the correct list. When a list is empty, a big chunk of memory is allocated using the standard allocator, subdivided in chunks of the right size, which are then placed in the appropriate free list.

The main advantages are two. The first is a decreased memory usage, because the custom slab allocator does not keep any additional management structure for each allocation; this also implies a reduction in CPU usage, as there are fewer memory structures to update during memory operations. The second is that, since we have one independent allocator per thread, there is no lock contention during allocations; this also contributes to reduce the CPU usage.

Of course there are also disadvantages. First of all, the allocator must be told the size of the memory chunk when freeing, in order to put it back in the right free list. Using chunk headers would have caused alignment problems or would have complicated the code even more, if correct allocation alignment had to be accounted for. This is an implementation detail, but passing the wrong value when freeing will have catastrophic results, which result in very elusive bugs. Another disadvantage is that the standard system libraries have some built-in checks to catch some possible memory corruption issues, which are of course absent in the custom slab allocator. And finally, tools like Valgrind[62] will not work with custom allocators. It can be seen that all the disadvan-

Table 3.1: Comparison of Bro, Tstat and HACkSAw. The best values for each column are in bold.

Tool	Detected requests	CPU [sec]	RAM [GB]	Replies w/o size	Relevant replies
Tstat	2 531 210	445	**0.3**	1 128 109	1 403 101
bro	**2 559 056**	8033	4.2	424 355	**2 134 701**
HACkSAw 0.2	2 426 391	368	5.8	328 465	2 097 926
HACkSAw 0.4	2 393 514	**235**	**0.3**	**269 311**	2 124 203

tages concern development and debugging, so to help debugging the slab allocator can be disabled at compile time.

The end result is that with the custom slab allocator there is roughly a 10% increase in the performance combined with a 10% reduction in memory consumption.

3.2.7 Performance

The final version of the tool can safely process two full-duplex 10Gpbs links for months with two cores and 20GB of RAM, when using DAG capture cards.

A comparison of the performance of the different tools is reported in Tab.3.1, using as benchmark the same one hour PCAP packet trace, which had been collected previously by other means. The table's columns report, respectively: the number of distinct HTTP requests; the total CPU time, cumulated across threads; the total amount of RAM used; the number of detected requests without a size; the number of detected requests with a size.

All tools detect approximately the same amount of requests; *bro* uses 10 times more memory and runs over 20 times slower than *Tstat*, whereas HACkSAw manages to be almost as accurate as *bro*, and as fast as *Tstat*. *Tstat* catches most of the transactions though it fails to match requests with the replies, and it fails to report the size of the object (at least when the Content-Length header is not within the first IP packet of the reply, which happens more than 30% of the cases in our benchmark). HACkSAw v.0.4 is the fastest and arguably very accurate, since it collects almost all the full transactions.

In the latest location, with two full-duplex 10Gigabit links, HACkSAw is using only two cores of the eight available. Per-core CPU usage varies from 35% during off-peak hours to 60% during peak hours. Total memory consumption stays under 20GB (thus 10GB per thread) after two weeks of analysis. The disk bandwidth to write the output logs is less than 1.5MB/s per thread. With our current hardware and additional capture cards we estimate that we could probably scale up to four or maybe even eight full duplex 10Gbps links.

3.3 Datasets

This section introduces the datasets collected using the tool, the statistics collected in the dataset, and some details on the dataset itself.

3.3.1 Statistics collected and overview

The tool collects many details about the TCP, UDP, HTTP and DNS transactions. The statistics used to compute cacheability are introduced here, for a full detailed list of all the collected statistics, see Appendix B.

Each detected HTTP request/reply transaction is recorded with a number of fields and associated statistics: time-stamp, user ID, object ID, Content-Length[63], actual transaction length over the wire, and range information due to HTTP `Range` requests.

The time-stamp is used for time-correlation and timeslot binning and it is in the Unix time format (number of seconds since January 1st 1970) with microsecond accuracy.

The user ID is the MAC address of the home router of the customer. Since customers get new IP addresses each time the home router reconnects, we needed a more stable identifier, and customers very rarely change their home router. For privacy reasons we actually used a salted hash of the MAC address.

The Object ID is the full URL of the requested object, concatenated with the ETAG header, if present. The ETAG is a header optionally returned with the reply, and it is used to indicate whether a given object has changed or not. It consists of a string, usually some kind of timestamp; equal values of ETAG for the same object indicate that the object hasn't changed in time. The ETAG is important for objects that change very frequently in time, like the front page of a news website. The final result is then hashed, both for efficiency reasons, since a 64 bit value is easier to handle than a potentially very long variable length string, and for privacy reasons, since URLs may contain sensitive information.

The Content-Length field is the object length optionally, but very frequently, advertised in the HTTP reply. In case of Range requests, the Content-Length is the length of the requested range, and not the whole object. In Chapter 5 we use this field to further disambiguate between different objects with the same URL.

The effective length is the number of bytes really transmitted on the wire. In most cases it is the same value as the Content-Length, but in case of interrupted or aborted downloads it will be a smaller value. This value is always present, but can be zero in case no content was transferred, like in case of redirects or errors.

The range information is used in Chapter 5 to properly reconstruct the size and amount of bytes transferred by separate Range requests for the same object, which are very common in video and audio streaming. The range information indicates the first and last byte of the requested range, and the total length of the object. This is the only place where the total object size is indicated in case of Range requests, since the Content-Length will be the size of the requested range (e.g. $last - first + 1$),

3.3.2 The datasets

Our methodology is based on deep packet inspection (DPI) of the HTTP protocol and therefore it does not apply to HTTPs, where all bytes transferred across a TCP connection are encrypted. We observed 15% of HTTPs web traffic in the first dataset, and 30% in the second; this statistic is expected to grow in the future, as HTTP 2.0 specifies encryption by default. In order to measure the amount of HTTPs traffic, we simply measured the amount of bytes transferred over ports 80 for HTTP and 443 for HTTPS. This is not a perfectly accurate way of measuring, because we are intentionally ignoring traffic on non-standard ports, but here we are not interested in an overly accurate figure, we only want to assess if the amount of unencrypted traffic is still a significant portion of the total traffic.

Considering the highly predominant usage of HTTP over HTTPs in our dataset, the results presented in this thesis are valuable to draw significant conclusions regarding cache performance at the network edge.

We collected several datasets, the ones used for scientific purposes are:

— the first one (`2014`) spans 42 days, from April 18th to May 30th 2014

— the second one (`2015-1`) spans 30 days, from January 10th to February 9th 2015

— the third one (`2015-6`) spans from June 11th to November 23th 2015, with two interruptions, for a total of 132 days.

Tab.3.2 reports a summary of these datasets in terms of the average daily and total number of objects and clients, comparing them to other datasets from other related works. HACkSAw version 0.2 was used to capture the first trace, while the much-improved version 0.4 was used to capture the third trace; the second trace was captured with version 0.3, a work in progress between the other two versions.

As indicated above in 3.1, the first dataset was collected in the edgemost location of the network, right above the OLT. The other two datasets were collected one step closer to the core, just below a DSLAM.

Table 3.2: Comparison of the datasets in this and related works. The largest values for each column are highlighted in bold.

Reference	Length	Clients	Requests	Distinct	HTTP	Savings	
				objects	traffic	requests	traffic
2015-6[83]	**132 days**	40k	**26.3G**	**8.9G**	**3.2PB**	52%	31%
daily average	-	*22k*	*194M*	*90.9M*	*23TB*		
2015-1[82]	30 days	30k	6.6G	2.5G	575TB	53%	40%
daily average	-	*24.5k*	*220M*	*102M*	*19TB*		
2014[79]	42 days	1.5k	369M	174M	37TB	42%	35%
daily average	-	*1.2k*	*8.6M*	*5M*	*881GB*		
[47]	14 days	20k	–	–	40TB	16–71%	9.5–28%
[48]	14 days	20k	–	–	42TB	–	–
[49]	1 day	200k	48M	7M	0.7TB	10–20%	13–40%
[31]	8 days	**1.8M**	7.7G	–	248TB	54%	41%
[30]	1 day		42M	–	12TB	16%	7%
[23]	1 day	–	6M	–	–	–	–

Table 3.2 shows a comparison of our datasets with the most notable datasets used in literature; the largest values are shown in bold, for our datasets a daily average is also presented.

Our first dataset has a very small user fanout (less than 2000 users), due to the positioning at the very edge of the network. This allows us to assess the effectiveness of a cache even with a small amount of distinct clients. Our second and third dataset have more typical user fanouts. Still, we do not come anywhere near [31], since that dataset was collected in the core, while we are positioned at the edge, inside the access network.

From a length perspective, our datasets are by far the longest. The third one, in particular, spans over 4 months. Such a long dataset allows us to observe the evolution of the cacheability and traffic reduction statistics in time over a long timespan; as we will illustrate in the next chapters, we observe a rather stable trend.

The sizes of the datasets themselves are huge, considering number of requests, number of objects and total amount of traffic. The sheer size of the datasets is both an advantage and a curse. A huge dataset will in general provide more credibility to any statistic calculated from it, with the downside of having to perform computations on extremely huge amounts of data. Our third dataset in particular is bigger than any of the other ones we observed in literature.

Also just considering the volume of the HTTP traffic, we can see that the the first dataset is on par with most other datasets, the second is twice as big as the next biggest one, and the third, with $3.2PB$, dwarfs all the other datasets combined. The sheer size of our datasets, with its tens of billions of HTTP transactions, posed a huge scalability problem for us; analysing all that information, especially the third dataset, forced us to look into big data solutions, as explained later in Chapter 5.

It is important to notice that the cacheability statistics in all our datasets are consistent, both with themselves and with the datasets used in literature.

Chapter 4

Cacheability analysis of real traffic

A common assumption to the evaluation of caching systems performance is to assume that content requests are generated under the IRM (Independent Reference Model), with a content popularity distribution following a Zipf law. The IRM considers all interactions independently from each other, therefore it completely disregards the temporal correlation between the requests.

Considerable measurement efforts have been devoted to provide input to this workload in static settings, while very few consider the catalogue dynamics over time. Characterisation of video catalogues has especially attracted significant attention, from YouTube crawling [54], to measurement at a campus [46] or ISP [28, 29], which show that IRM assumption may lead to a significant under-estimation of the achievable caching gains.

In this chapter we will introduce the statistics and metrics used in this thesis. The statistics all refer to a time interval, so after introducing the statistics we will perform some timescale analysis to find reasonable time intervals to calculate the statistics on. The statistics will then be used to estimate the size of real caches, and LRU cache simulations are then performed using the estimated cache sizes. We will see that a sizeable share of traffic can potentially be cached.

4.1 Key Statistics

We now introduce the statistics that will be used throughout this thesis. They are all applicable on a time interval, and provide some metric (an upper bound) of the amount of traffic or requests that can be cached. Figure 4.1 provides a visual representation.

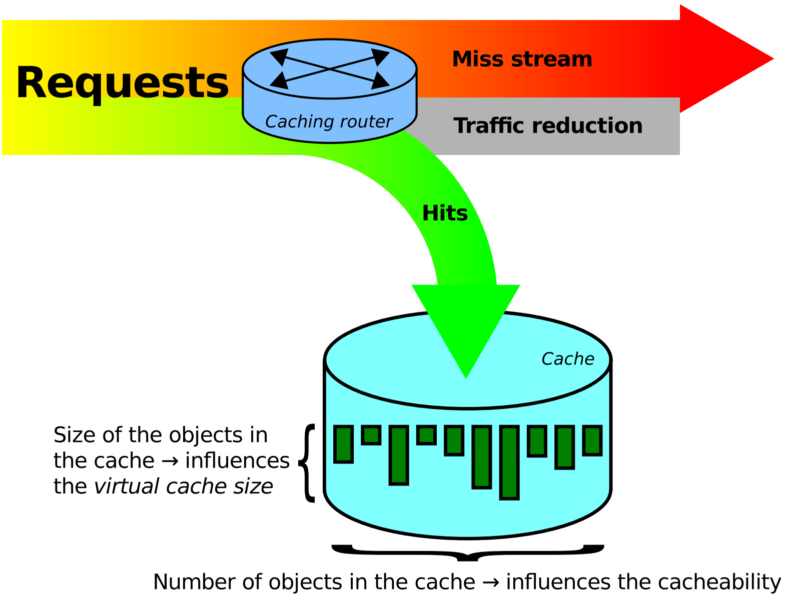

Requests
Caching router
Miss stream
Traffic reduction
Hits
Cache
Size of the objects in the cache → influences the *virtual cache size*
Number of objects in the cache → influences the cacheability

Figure 4.1: Illustration of the analytics associated to ISP caching: *object cacheability*, *traffic reduction* and *virtual cache size*.

Let us define as *catalogue* C the set of distinct objects requested in a given time interval, and denote with $N_o = |C|$ its size in terms of the number of distinct objects. While we do not make any assumption, we remark that the catalogue is generally very large and rather typically most of those objects are requested only once in the time interval.

We now define as *cacheability* the fraction of requests for objects (or parts thereof) requested more than once in a given time interval. This index is fairly commonly used [47, 64, 65][79] to find the upper bound of the expected benefits of reactive caching. The first request of an object is not considered cacheable[1], whereas all its subsequent requests in the same time interval are. Specifically, cacheability is an upper bound since deterministically assuming all subsequent request to generate a hit implicitly assumes the cache to be large enough to avoid content eviction due to cache size limit. Considering for the time being full-object requests for the sake of simplicity, and denoting with N_{hits} the number of potential cache hits, with N_r the number of requests observed in the given time interval, and with N_o the catalogue size defined above, cacheability can be defined as:

$$\frac{N_{hits}}{N_r} = \frac{N_r - N_o}{N_r} = 1 - \frac{N_o}{N_r}$$

[1] It would generate a cache hit in case of proactive prefetching, as opposite to reactive caching settings considered here

Notice that cacheability is an object-wise metric: we also need a byte-wise metric that explicitly takes into account the volume of objects. It's in fact possible that small objects are more cacheable than bigger objects (and we will show later in 5.3 that this is indeed the case), so it is also important to estimate the actual amount of traffic potentially saved by caching. We therefore define *traffic reduction* as an index measuring the maximum amount of traffic that can be potentially saved across a link over a given time interval as a consequence of content cacheability. Denoting with R the total traffic, with R_u the uncacheable traffic, and R_c the cacheable traffic (all measured in bytes), traffic reduction can be defined as:

$$\frac{R_c}{R} = \frac{R - R_u}{R} = 1 - \frac{R_u}{R}$$

Finally, we observe that cacheability and traffic reductions are an *upper bound* of the achievable performance. We therefore need an estimate of the cache size required to achieve such performance, and denote this index as *virtual cache size.* We observe that, in case the arrival process would strictly order all requests for the same content (i.e. all requests for an object i arrive in sequence before requests for object $i + 1$) then the virtual cache size would simple amount to the largest object in the catalogue. This estimate is however very optimistic, so that we prefer a conservative approach that provides an upper-bound to the cache size: this bound is suitable when requests for all objects are intermingled, so that there is no temporal correlation between requests (i.e. requests arrive in random order with a rate proportional to the popularity of each object in the catalogue). In this latter case, the *virtual cache size* is defined as the sum of the observed size V_o of the cacheable objects o:

$$\sum_{o \in cacheable objects} V_o$$

where cacheable means that the object o has been seen at least twice in the timeslot and the size V_o is the number of distinct bytes of content observed for the object o.

Notice that these statistics are relative to a specific timeslot, and timeslots are independent. So slicing the traces in timeslots disregards the temporal locality of requests across timeslot boundaries. That's also why finding the optimal timeslot size is important.

4.2 Timescale analysis and content popularity

Traffic characterisation is an essential prerequisite of traffic engineering. Network dimensioning and upgrading lie upon the knowledge of the relation between three entities: traffic demand, network capacity and quality of service. What makes traffic characterisation a difficult task is the stochastic nature of Internet traffic, complex to synthesise via simple models.

In literature, a wide range of models exists, varying model abstraction and related complexity according to a more microscopic or macroscopic analysis of network dynamics. In this thesis we avoid a detailed representation of a network of caches, which turns out to be very difficult to represent analytically even for a tandem cache and simple workloads [66, 67, 68]. We rather prefer a simplified characterisation of web traffic, based on key system properties and applicable to general in-network caching systems. Such model abstraction, assuming an independent reference model (IRM), might be leveraged for the dimensioning of a micro CDN. The key factors impacting the performance of an in-network caching system and that our model takes into account are:

— the timescale at which content popularity may be approximated by an independent reference model (IRM) (Sec.4.2.1);

— the content popularity at the given timescale (Sec.4.2.2).

From this characterisation, we later infer in Sec.4.3.2 the minimum useful amount of memory to embed in the home network and in edge routers in the micro CDN architecture.

4.2.1 Timescale analysis

Before presenting observations from the network probe, we use a simple explanatory model to show that measuring content popularity at a timescale where the IRM assumption does not hold may lead to wrong predictions in terms of memory requirements (e.g. over a large time window).

We divide the time axis in windows of size $T > 0$, $W_i = [iT, iT + T)$, and assume that, in each time window W_i, objects in content catalogue A_i are requested following a Poisson process of rate λ, with $A_i \cap A_j = \emptyset$ for all $i, j : i \neq j$. The average object size is σ bytes. A_i is Zipf distributed with parameters α, N, i.e. a content item of rank k is requested with probability $q_k = ck^{-\alpha}$, $k \in \{1, \ldots, N\}$, $|A_i| = N$.

By using the modelling framework developed in [26] for an LRU cache of size x in bytes, we know that if $T >> x^{\alpha} g$, with $1/g = \lambda c \sigma^{\alpha} \Gamma(1-1/\alpha)^{\alpha}$ the cache miss probability for an object of rank k tends to $\exp\{-\lambda q_k g x^{\alpha}\}$.

However, if one estimates the content popularity as the time average across m contiguous time windows, the estimated miss probability would be $\exp\{-\lambda q_k g(x/m)^\alpha\}$. Indeed, the right cache performance measured across m contiguous time windows of size T is still $\exp\{-\lambda q_k g x^\alpha\}$ resulting in an overestimation factor m of the required memory, for the same miss ratio.

In this section we estimate the timescale over which the IRM model can be used to estimate cache performance without using complex measurement-based models, e.g. [28],[29].

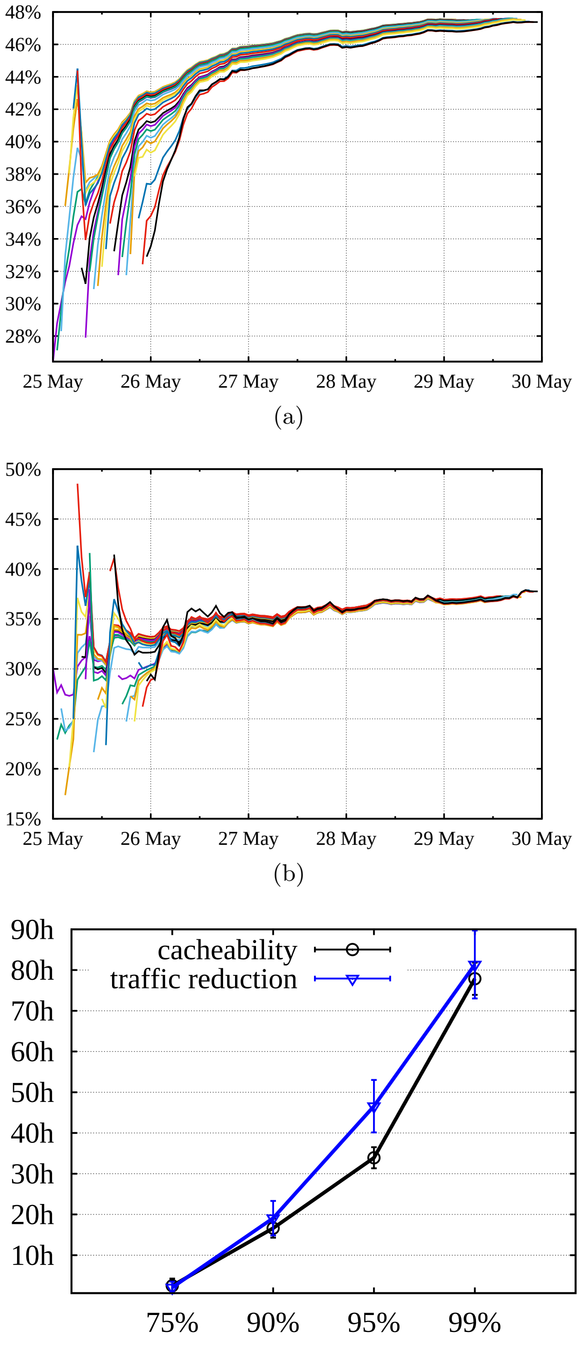
Observation 4.2.1. *In order to exploit a IRM cache network model for system dimensioning, one needs to estimate the smallest timescale, referred to as "cutoff" timescale at which the IRM assumption holds. As a consequence, above the cutoff timescale, every new content request gives a negligible contribution to catalogue inference.*

In Fig.4.2(a),(b) we plot cacheability and traffic reduction as computed over our first dataset (2014) at different time windows: from one hour to an entire contiguous week at incremental steps of one hour. The statistics are also computed starting at different time instants, delayed by one hour each. We observe that the two statistics have a cutoff scale above which they reach a plateau. In Fig.4.2(c), we report the time required for the cacheability to attain percentiles (namely, the 75%, 90%, 95% and 99%) of the long term value: it can be seen that 90% of the traffic reduction are attained in less than 24 hours, irrespectively of the start time.

Observation 4.2.2. *The cutoff scale is hard to be measured as it changes on a daily basis as a function of many factors that cannot be rigorously quantified. However, we observe that for practical purposes aggregate web traffic would benefit from caching no more than a daily content catalogue.*

In Fig.4.2(a),(b) we also observe that the cacheability stabilises at about 47% while traffic reduction amounts to almost 37%. These values provide a first rough estimation of the opportunities to cache data currently available within the ISP network at relatively low user fan-out. While the statistics just presented provide insights on the potential gains achievable by caching a daily content catalogue, we now investigate temporal evolution of the catalogue.

To this aim, we introduce a measure of auto similarity based on the *Jaccard coefficient*, that indicates the proportion of objects in common between two given sets: $J(A,B) = \frac{|A \cap B|}{|A \cup B|}$ (If $A = B = \emptyset$ then $J(A,B) \triangleq 1$. Clearly, $0 \leq J(A,B) \leq 1$. We then



(a)

(b)

(c)

Figure 4.2: Cumulative cacheability (a) and traffic reduction (b), starting from different hours and for various timespans. (c) Hours needed to reach some percentiles of the cacheability and traffic reduction plateaus, with standard deviations.

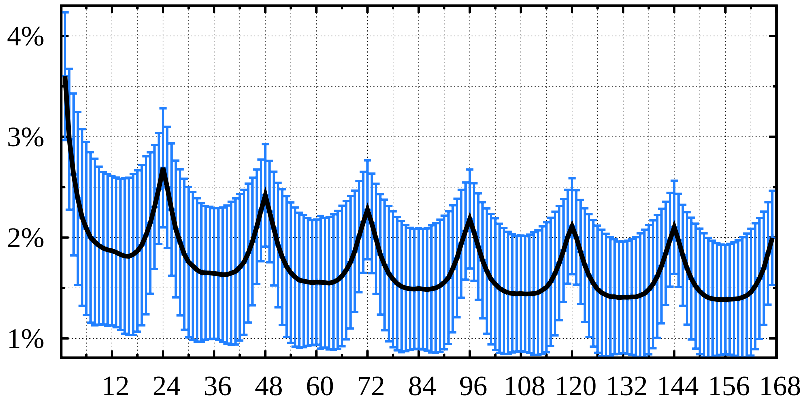


Figure 4.3: Jaccard auto correlation function $\mathcal{R}_{\Delta T}(k)$.

define the Jaccard auto correlation function as

$$\mathcal{R}_{\Delta T}(k) = \frac{1}{n} \sum_{i,j:|i-j|=k}^{n} J(C_{i\Delta T}, C_{j\Delta T})$$

being $C_{i\Delta T}$ the content catalogue measured over the time window $i\Delta T$. Fig.4.3 shows $\mathcal{R}_{\Delta T}(k)$ for $k = \{0, \ldots, 168\}$, ΔT = 1hour, during one working week in May 2014 (showing standard deviation as error bars). An interesting conclusion can be drawn.

Observation 4.2.3. *The catalogue is weakly auto-correlated, as $\mathcal{R}_{\Delta T}(k)$ falls from 100% to less than 5% and it completely regenerates asymptotically, as $\mathcal{R}_{\Delta T}(k) \to 0$ when $k \to \infty$. A periodic component with period of about 24 hours is also present as a result of users' daily routine.*

Finally, we show that the catalogue exhibits a clear night/day effect, Fig.4.4 reports $J(C_{k_0\Delta T}, C_{(k_0+k)\Delta T})$, with $\Delta T = 1$hour and $k_0 < k \leq 72$, for multiple $k_0 = \{0, 2, 4, 6, 8\}$. The figure shows that the catalogue has different properties during off peak hours ($k_0 = \{0, 2, 4\}$) than peak hours ($k_0 = \{6, 8\}$). Off-peak hours are characterised by content items that unlikely appears again in the future, while on-peak content items show periodic components of 24 hours.

4.2.2 Content popularity estimation

According to previous observations, the timescale of interest turns out to be approximately defined by removing the night period, i.e. the off peak phase. This may be a complex task as the off peak phase changes on a daily basis. Nevertheless, we observe that the off peak phase has statistically weak impact on the overall distribution as it

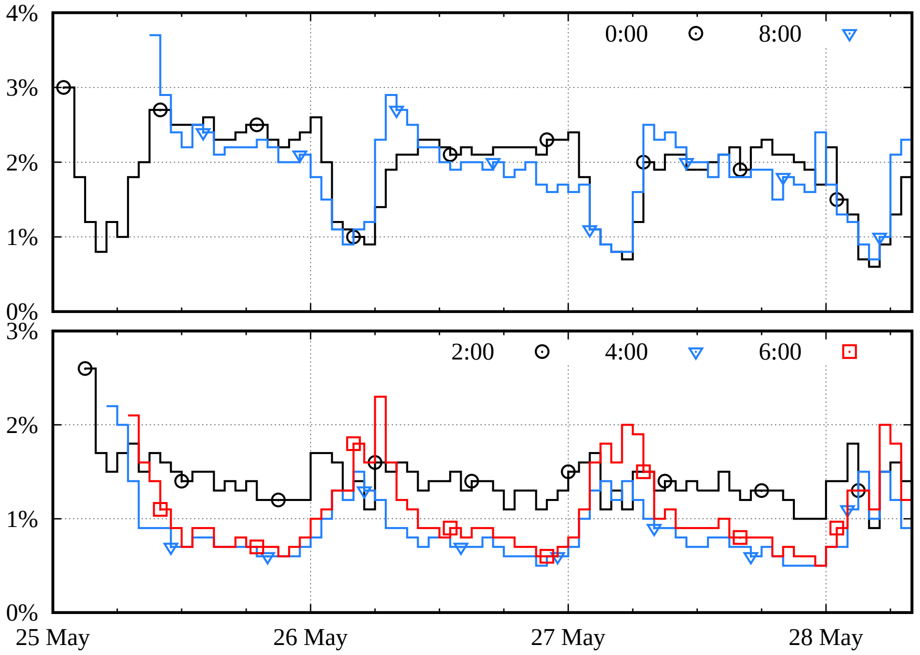


Figure 4.4: Jaccard coefficient, $J(C_{k_0\Delta T}, C_{(k_0+k)\Delta T})$, $k_0 = 0, 2, 4, 6, 8$.

carries samples in the tail of the popularity distribution at low rate, so that 24 hours can be used as good approximation timescale. We now present a model of content popularity estimated over 24 hours.

We test the empirical popularity against various models and find the discrete Weibull to be the best fit with shape around 0.24 (long tailed). In Fig.4.5, we report the empirical popularity distribution with corresponding 95% confidence bands (see [69] for similar analysis). We also plot, in red, the model fit to the available sample with 95% confidence bands. By means of extensive tests on this model we assess accuracy over all 24 hours samples on our data set. It follows that (i) for the *tail* of the distribution, a simple discrete Weibull passes a χ^2 goodness of fit test [70], with p-values exceeding 5% significance level, (ii) the good model for the *entire distribution* turns out to be trimodal with three components: a discrete Weibull for the *head* of the distribution, a Zipf for the *waist* and, a Weibull again for the *tail*, i.e. $f(k) =$

$$\begin{cases} \phi_1 \dfrac{\beta_1}{\lambda_1}\left(\dfrac{k}{\lambda_1}\right)^{\beta_1-1} e^{-(k/\lambda_1)^{\beta_1}} & k < k_1 \\ \dfrac{\phi_2}{k^{\alpha_2}} & k \in [k_1, k_2] \\ \phi_3 \dfrac{\beta_3}{\lambda_3}\left(\dfrac{k}{\lambda_3}\right)^{\beta_3-1} e^{-(k/\lambda_3)^{\beta_3}} & k > k_2 \end{cases}$$

with parameters $\lambda_1, \beta_1, \alpha_2, \lambda_3, \beta_3, \phi_1, \phi_2, \phi_3 \in \mathbb{R}^+$; $k_1, k_2 \in \mathbb{N}$. The parameters have been estimated by using standard maximum likelihood (ML) applied to the piecewise function $f(k)$. The set of parameters of each piece of $f(k)$ is estimated independently to the others in order to fix the shapes exponents $\beta_1, \alpha_2, \beta_3$ and the scale factors λ_1, λ_3. An ML estimator is not available for the entire distribution and we therefore use the method of moments (MM) to determine ϕ_1, ϕ_2, ϕ_3. The procedure can be iterated to obtain a better estimation by running ML first and MM afterwards. In our samples after four iterations the parameters stabilise to stationary values that we have reported for one day in Fig.4.5 where $\beta_1 = 0.5$, $\alpha_2 = 0.83$, $\beta_3 = 0.24$.

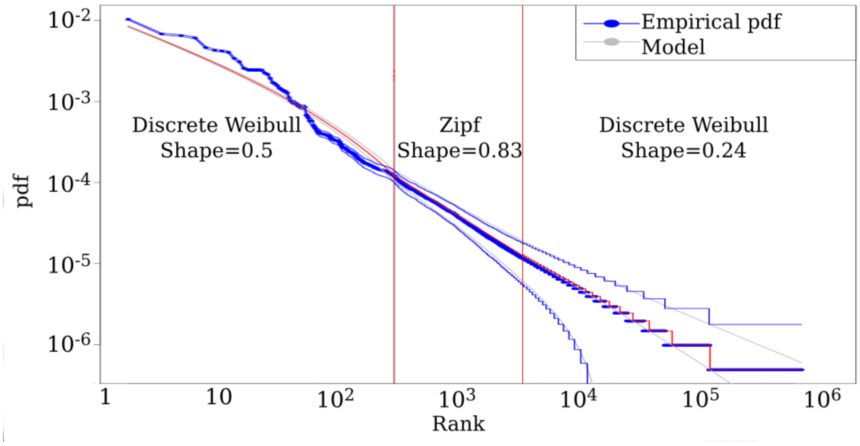


Figure 4.5: Empirical web content popularity and model fitting with corresponding confidence bands.

It is not uncommon to confuse a heavy-tailed distributions (e.g. Zipf) with a long-tailed one (e.g. Weibull). The most naive techniques based on the analysis of the linear regression of the loglog plots are known to be error prone. More powerful techniques to analyse extreme values of a distribution are based on the use of the Hill statistics [71],[72]. It is possible to build a hypothesis test based on the Hill statistic $H_n = 1/k \sum_1^k \log \frac{X_{(i)}}{X_{(k+1)}})$

for $k = 1, \dots, n-1$ being n the size of the sample and $\{X_{(i)}\}$ the ordered sample. The null hypothesis is $H_n \to \alpha$ to identify a Zipf tail and $H_n \to 0$ to identify a Weibull tail (see [73]). We have applied the test based on the Hill statistic on the popularity function of a month of data and we have found that the test $H_n \to \alpha$ is always rejected. The test $H_n \to 0$ is however never rejected. In Fig.4.6 we report the Hill statistic for three data samples: one is composed of quantiles simulated from a Zipf distribution with shape equal to 0.83 (a recurrent value in the literature that our analysis attributes only to the waist of the distribution, see Fig.4.5); the second sample is obtained by simulating a discrete Weibull distribution with shape equal to 0.24 (as obtained from our data). The third sample reported in the figure is obtained from our network measurements. All samples have the same size. From Fig.4.6 we observe the convergence properties of the different Hill statistics that would exclude a Zipf tail for content popularity.

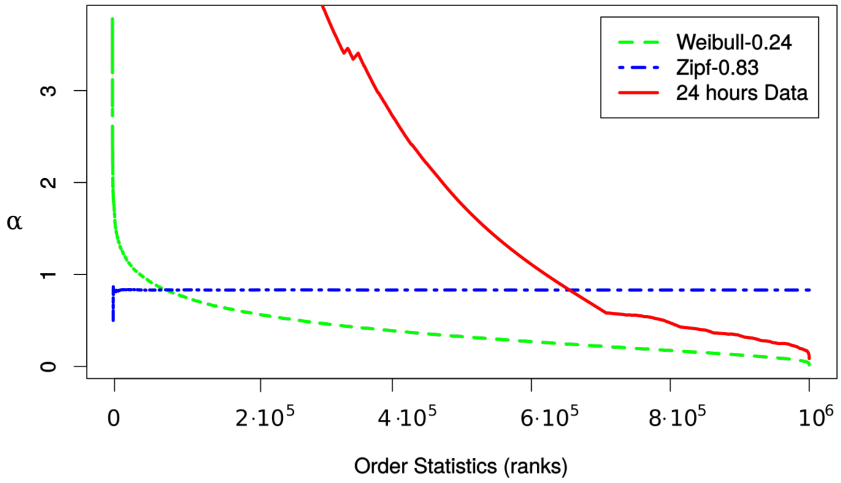


Figure 4.6: Hill plot comparison: Zipf-0.83, Weibull-0.24 and a 24 hours sample of data.

Interesting conclusions can be drawn from our popularity characterisation. In literature, the majority of analytical models assume a Zipf distribution with shape $\alpha < 1$ for the entire distribution. Remark that, if the same Zipf law characterising the waist is prolonged all over the support of the distribution, a finite support, corresponding to a content catalogue estimation, must be imposed. We recall that the miss probability of a cache of size x employing the LFU[2] replacement policy is given by $m(x) = P\{R > x\} = 1 - \sum_{k=1}^{x} q_k$. If we assume Zipf popularity with shape $\alpha < 1$ than $m(x) = 1-(x/N)^{1-\alpha}$. If we assume Zipf popularity with shape $\alpha > 1$ than $m(x) \sim x^{1-\alpha}$.

[2] Least Frequently Used

For a Weibull distribution $m(x) = e^{(x/\lambda)^\beta}$.

Cache performance would then depend on the ratio between cache and content catalogue size, while it is a function of cache size only under the more precise Weibull tail fit that we made. Moreover, the cardinality of the catalogue, N, is estimated with unbounded confidence intervals (see Fig.4.5), whereas all Weibull's parameters can be estimated with arbitrary low error, by increasing the size of the sample. As a consequence, an overestimation of the catalogue size by a given factor under the all-Zipf model would lead to memory over-sizing of the same factor for a given target miss ratio. Conversely, the miss ratio under Weibull requests [74], e.g. of an LRU cache, can be estimated with arbitrary precision by increasing the size of the sample to estimate the popularity law. Hence, we derive the following.

Observation 4.2.4. *Accurate content popularity characterisation is fundamental to drive a correct cache system dimensioning. Approximate models based on (i) all-Zipf assumption, (ii) possibly fit over long time scales, coupled to (iii) IRM model assumptions, may lead to excessively conservative results, and ultimately to misguided system design choices.*

4.3 Simulations

In the following, we present some statistics and some realistic simulations driven by our dataset. The goal is to (i) evaluate the amount of memory required within the backhaul to absorb cacheable traffic and (ii) to assess the accuracy of the model introduced in previous section for the dimensioning of a micro CDN system.

4.3.1 Scenarios

We simulate three different scenarios, in order to asses the impact of a cache in different points of the network.

Cache at vantage point only (OLT). This simulates a transparent proxy, placed as close as possible to the users, but still in the network of the operator. This is a rather simple setup, with only one cache. The challenges here are the speed, since many clients would hit the cache, and the encapsulation, since the packets would still be encapsulated in the access L2 protocol (e.g. PPPoE).

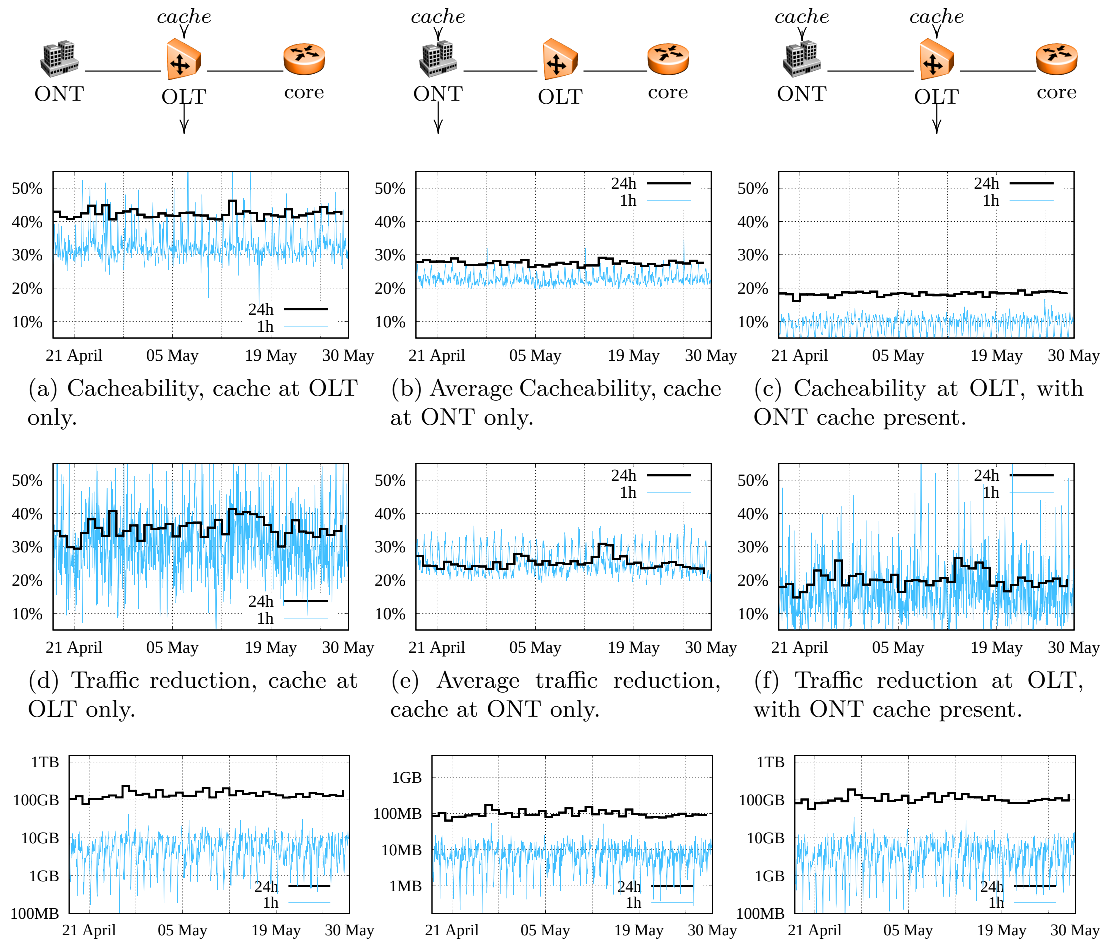
Caches at the users only (ONT). This simulates a transparent proxy installed on the home routers of the clients. This is also a simple setup, although it involves a significant amount of caches. The challenges here are (i) in the sizing of the caches, since some heavy users won't benefit from the relatively small cache on their router, and (ii) deploying and maintaining thousands to millions of transparent proxies on the routers of the clients.

Caches placed both at the vantage point and at the users (OLT and ONT). This simulates either a network of transparent proxies, or a deployment of CCN in a real network, where each node in the network is also a cache. In case of transparent proxies, some challenges remain (deployment, encapsulation), while other go away or are less relevant (cache sizing, since there is now a big cache above the clients, and line speed, since now fewer requests make it to the OLT cache). In case of a CCN deployment, there are none of the challenges, because caching is built-in.

4.3.2 Cacheability, Traffic reduction and Virtual cache size

We now analyse cacheability, traffic reduction and virtual cache size, as introduced in Sec. 4.1. We simulate the three scenarios as follows: in the *first scenario* all statistics are simply calculated on the whole dataset; in the *second scenario* all statistics are first calculated per-client, and then the mean average of all clients is then calculated. The clients are identified by MAC address, as already explained in Sec. 3.3; in the *third scenario* object cacheability is calculated per-client, like in the previous scenario. Per-client cacheable requests are then filtered out, all the remaining requests are put together, and all the statistics are then calculated on the resulting dataset. This simulates a perfect cache at each client plus a cache at the OLT. The performance of the cache at the clients is not analysed in this scenario because it would obviously be the same as in the previous scenario.

Fig.4.7 plots cacheability (top row), traffic reduction (middle row) and virtual cache size (bottom row), over more than one month in 2014, over hourly and daily timescales. Different scenarios are arranged by columns: namely OLT-only caches (first scenario, left column), ONT-only caches (second scenario, middle column) or OLT+ONT caches (third scenario, right column). The OLT-caching scenario in the left column of Fig.4.7(a),(d),(g) is striking: with a little more than 100GB of memory, 35% of average traffic can be saved for a user fan-out equal to 2048. In the ONT-caching scenario, only duplicated requests coming from the same users are filtered by the cache. In that case, an average memory size of about 100MB per user (thus adding up to 200GB in total, given the user fan-out) reduces user traffic by 25%, corresponding to a same level of load reduction in the GPON access. Finally, the last column of figures, Fig.4.7(c),(f),(i), shows that employing caches at both OLT and ONT level, the ISP network would benefit from 25% load reduction in the GPON and 35% on back-haul links, while improving the latency for all users. We



(a) Cacheability, cache at OLT only.

(b) Average Cacheability, cache at ONT only.

(c) Cacheability at OLT, with ONT cache present.

(d) Traffic reduction, cache at OLT only.

(e) Average traffic reduction, cache at ONT only.

(f) Traffic reduction at OLT, with ONT cache present.

(g) Virtual cache size, cache at OLT only.

(h) Average virtual cache size, cache at ONT only.

(i) Virtual cache size at OLT, with ONT cache present.

Figure 4.7: Time evolution of hourly and daily statistics; by rows cacheability, traffic reduction and virtual cache size are reported respectively in the three cases: (i) cache at OLT only, (ii) cache at ONT only and (iii) cache at OLT and ONT.

can thus conclude:

Observation 4.3.1. *Due to temporal correlation of requests, sizeable traffic reduction (in both the GPON and back-haul), can be achieved via deployment of caches of relatively modest size (estimated through the virtual cache size).*

We now briefly compare the statistics above with the ones from the second dataset. The probe is now at the BAS (see Fig.3.1), therefore closer to the core of the network; the user fan-out is significantly higher than in the first dataset (about 30 000 unique users). Our interest in not to repeat the whole analysis, but rather to see whether the cacheability and traffic reduction are compatible with the results shown above for the first dataset.

The averages of the daily cacheability and traffic reduction metrics, calculated over the whole dataset are respectively 53.6% and 40.2%. Notice that, considering only Web connections (TCP port 80 or 443), 35.5% of the connections and 30.4% of the traffic was SSL: while this represents a noticeable increase compared to the old dataset, it does not invalidate the methods used, because the remaining unencrypted (and therefore analysable) traffic is still a meaningful share of the total. We thus gather:

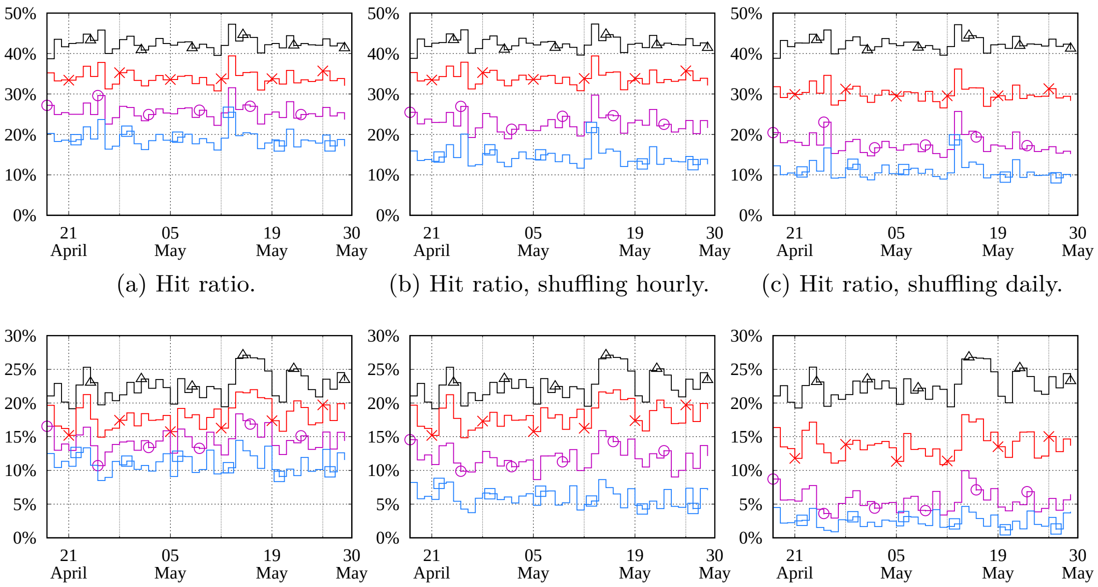
Observation 4.3.2. *Cacheability and traffic reduction statistics are in line with those presented in the previous sections, and in particular the difference between cacheability and traffic reduction is preserved. The higher values are a direct consequence of the higher level of aggregation achieved in the new location – which explicitly confirms the conservativeness of the results gathered in the previous sections.*

4.3.3 LRU cache simulation

The first set of simulations is based on LRU caches, set up in accordance with the three scenarios, and driven by real users' requests. We simulate LRU caches with different sizes and measure the average hit ratio and the potential traffic savings over a 1 day timescale. The LRU caches simulates a transparent cache: a web object is stored in chunks, so that in case of a cache miss, only the missing chunk is requested. A following request for a bigger part of an object partially present in cache generates another miss, but only for the missing part of the object.

For each scenario, the statistics are calculated from the real traffic, and from two additional artificial request streams generated by uniformly shuffling the real requests in 1 hour and 1 day timeslots. Request shuffling is useful to remove time correlation, which has huge impact on cache performance as already discussed in Sec.4.2: in particular, shuffling produces a workload closer to that of IRM model.

We start by considering cache sizes of 1GB, 10GB, 100GB and 1TB at the OLT (first scenario), and use standard LRU replacement. Its performance is reported in Fig.4.8(a),(d) and compared with the two additional systems obtained by shuffling all the requests on a hourly and daily basis, reported in Fig.4.8(b),(e) and Fig.4.8(c),(f) respectively. It can be seen that, while the hit ratio in the best case follows the cacheability, the saved traffic is significantly less. This is because the statistic of traffic reduction assumes that cacheable content will be prefetched and available before the first request is sent, whereas when simulating an LRU cache no prefetching takes place.



(a) Hit ratio.

(b) Hit ratio, shuffling hourly.

(c) Hit ratio, shuffling daily.

(d) Saved traffic.

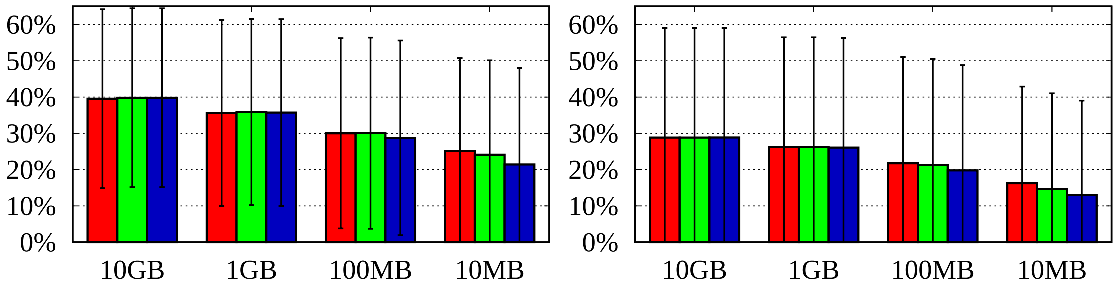
(e) Saved traffic, shuffling hourly.

(f) Saved traffic, shuffling daily.

Figure 4.8: LRU cache simulations, behaviour with one-day timeslots. First scenario: cache at OLT only. It can be seen that the saved traffic is less than the traffic reduction, as explained in section 4.3.3. Cache sizes: △: 1TB, ×: 100GB, ○: 10GB, □: 1GB.

Similar results are shown in Fig.4.9 for the second scenario, where the average and standard deviation for hit ratio and saved traffic for all clients are shown. In this case the caches have sizes of 10MB, 100MB, 1GB and 10GB. The standard deviation shows that the actual behaviour of the clients is far from being homogeneous: while some clients exhibit very redundant request patterns (over 60% of cacheable traffic), patterns of others clients are completely uncacheable. Traffic reduction shows an even more drastic variance compared to cacheability. We also notice that the values reported are higher than the values for cacheability and traffic reduction. This is due to the excessively large caches used in the second scenario of the LRU simulations, combined with the fact that the LRU caches are not emptied at timeslot boundaries.

Finally Tab.4.1 shows the results for the third scenario; since there are now two independent caches with varying size, only a global average is presented. The numbers indicate only the performance of the OLT cache because the ONT caches will behave exactly as in the second scenario; since the OLT cache only processes requests from ONT, the performance of the ONT cache itself is not influenced by the presence of the OLT cache.



(a) Hit ratio

(b) Saved traffic

Figure 4.9: LRU cache simulations, behaviour with one-day timeslots. Second scenario: cache at clients. Average and standard deviation of the hit ratio and saved traffic for all clients, for different ONT cache sizes. Section 4.3.3 explains why these values are higher than the values of cacheability and traffic reduction. Actual traffic:🟥; 1h shuffled traffic:🟩; 24h shuffled traffic:🟦.

From the simulations we see that if the cache is big enough (1TB for the OLT, 10GB for the ONT), time correlations do not have impact on performance. For medium or small caches, instead, the performance in presence of temporal locality of requests is more than halved in the first scenario; in the second scenario the performance is also impacted, but not as badly as in the first scenario.

The third scenario, instead, shows some numbers that are at first sight counter-intuitive, but are easily explained considering that the performance of the ONT and OLT caches is inversely related. Whenever the lower level caches are too small or ineffective, the upper level will receive more cacheable traffic, and its performance will therefore increase. In particular we observe that both smaller ONT caches and shuffling cause an improvement in the OLT performance. The presence of a relevant amount of cacheable traffic at the OLT even in presence of big ONT caches shows that there is indeed a consistent amount of shared traffic between different clients; even with huge 10GB caches at the ONT, more than 8% of the traffic is still cacheable at the OLT. We can summarise the above with the following observation:

Observation 4.3.3. *A considerable part of the traffic is cacheable; a large part of the overall cacheability is due to repeated requests by the clients, but a considerable fraction of the requests are still shared across different clients. Therefore it makes sense to place caches at both the ONT and OLT. Moreover, due to the large variance in the performance of ONT caches, a cache at the OLT is needed anyway to catch the objects requested by users whose traffic pattern is not cacheable at the ONT. Specifically, ONT cache is beneficial to reduce the load in the access part (which is often a shared medium like fibre*

	1GB	10GB	100GB	1TB
10MB	3.2	8.9	18.2	28.7
100MB	2.2	6.7	15.2	25.6
1GB	1.6	4.8	11.8	20.7
10GB	1.5	3.6	9.2	16.6

(a) Hit ratio (%).

	1GB	10GB	100GB	1TB
10MB	2.5	9.1	20.2	30.5
100MB	1.5	6.2	15.7	26.0
1GB	0.9	4.2	11.7	20.7
10GB	0.7	3.1	9.1	16.5

(b) Hit ratio (%), shuffling hourly.

	1GB	10GB	100GB	1TB
10MB	2.4	7.4	19.3	33.4
100MB	1.2	5.1	14.5	28.2
1GB	0.5	3.0	10.2	21.1
10GB	0.3	1.8	7.6	16.4

(c) Hit ratio (%), shuffling daily.

	1GB	10GB	100GB	1TB
10MB	4.3	7.4	11.3	16.1
100MB	2.0	4.1	7.9	12.8
1GB	1.6	3.1	6.3	10.7
10GB	1.6	2.7	4.8	8.1

(d) Saved traffic (%).

	1GB	10GB	100GB	1TB
10MB	2.0	7.6	13.6	18.3
100MB	0.6	3.8	9.2	14.0
1GB	0.4	2.4	6.5	10.9
10GB	0.4	2.0	4.9	8.2

(e) Saved traffic (%), shuffling hourly.

	1GB	10GB	100GB	1TB
10MB	0.6	3.0	10.9	20.0
100MB	0.2	1.5	7.1	16.4
1GB	0.2	1.1	4.4	11.7
10GB	0.1	0.9	3.4	8.2

(f) Saved traffic (%), shuffling daily.

Table 4.1: LRU cache simulations, third scenario: cache system with cache at clients and at vantage point; averages of daily values. The behaviour of the ONT caches in the third scenario is exactly the same as in the second scenario. Client caches of sizes 10MB to 10GB; vantage point cache sizes of 1GB to 1TB. The apparently counter-intuitive values are explained at the end of section 4.3.3.

in our case or cable), while ONT cache is useful to relieve the load in the upstream backhaul network (which is a precious resource as well).

4.4 Object identification

Cacheability and traffic reduction both heavily depend on a correct identification of the transferred objects in order to determine if a given object was requested more than once. This section thus aims to explain the systems used to identify and distinguish the objects. Especially, our aim is to assess the robustness of the previous findings against different, increasingly sophisticated, object identification methods. This is important not only from a methodological point of view, but also because, to our knowledge, this kind of analysis is novel.

Before introducing the technique we consider, we need to introduce protocol details to illustrate the design space for object identification.

4.4.1 HTTP Transactions and Headers

An HTTP transaction consists of a request from a client and its reply from the server. The request consists of a method (usually GET), a resource, and the HTTP version supported by the client, followed by some headers. The reply consists of the

HTTP version supported by the server, a numerical code indicating the outcome of the request, and an optional description, followed by some headers. All headers have the same structure: they are simple key-value associations. Each HTTP transaction has several headers and properties that can be used to identify objects, the most obvious one is the resource part of the request, which, together with the `Host` header constitutes the URL.

A useful, optional, information is carried by the `ETAG` header. ETAG consists of a string that uniquely identifies the specific version of the requested content so that, when present, it gives a clear indication if two transactions for the same URL are about the same underlying object. ETAG is generally used for caching purposes: the client can perform ETAG-based conditional GETs, and the HTTP standard mandates that the same URL with the same ETAG should correspond to the same content. Using the ETAG therefore allows to distinguish between different versions of the same page (e.g. news websites). Table 4.2 shows that more than 30% of the observed requests have an ETAG header.

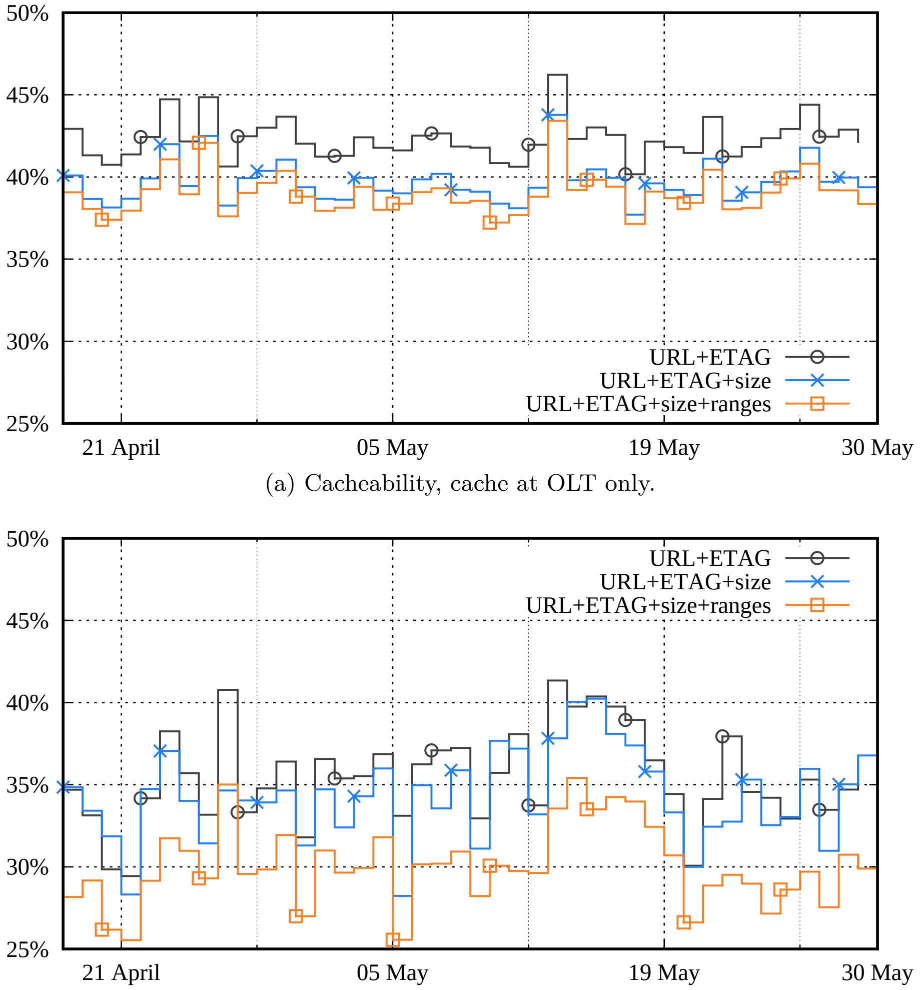
Table 4.2: Statistics of headers useful for content identification (top) and comparison of the first two identification methods (bottom)

HTTP requests	Occurrence	Penetration
with ETAG	116 819 069	31.64%
with cookies	191 075 062	51.75%
`Range` requests	6 869 802	1.86%

Identification method	Distinct objects
URL+ETAG	174 283 000
URL+ETAG+size	184 122 000

HTTP range requests, instead, complicate the matters. A client can request only a given byte range of the content, using the `Range` header, and if the server supports range requests, a `206 Partial Content` reply is issued instead of the usual `200 OK`, the `Content-Length` header indicates at this point the length of the requested range. An additional `Content-Range` header is also issued in the reply, indicating the byte range and the total length of the object. If the server does not support range requests, it will reply with the whole object, disregarding the `Range` header. Range requests are used mostly in three cases: to transfer large amounts of data in a safe and verifiable way, to perform media streaming, and to resume interrupted downloads. The first case relates to big downloads, mostly performed through specialised clients, like system updates, where the client can verify and re-request corrupted pieces of the file, without having to download all of it again. The second case is for media streaming (audio and/or video): the client requests pieces of the media as the media itself is being played to the user, and as users might skip to random parts of the video, they could thus not be downloading the whole object. The third case is the simple case where a download was interrupted for any reason (e.g. poor Wi-Fi coverage, software crash, battery discharged, etc), and then resumed. In most (but not all) cases the client will only attempt to download the missing parts. We observed all of the above behaviours in our dataset; moreover the average size of all HTTP transactions observed in our dataset is 110KB, whereas the average size of transactions with range requests is 1161KB, which is over 10x times bigger and is in line with the explanations above. Therefore, despite being less than 2% of the requests, range requests can potentially represent 20% of the traffic, thus our interest in assessing their real impact.

There are several other headers that can potentially influence the content returned by the server for the same URL – including cookies, language, user-agent and referrer headers. At each transaction, servers may request cookies to be stored by the client (`Set-Cookie` header), which the client will subsequently send back to the server (`Cookie` header) when requesting further objects. Cookies are ubiquitously used for tracking users and potentially serve personalised content – yet different cookies do not automatically imply different content. Clients can specify a list of languages they like (`Accept-Language` header), and the server can specify which language is being used in the reply (`Content-Language`). This is used to get the content in a language the user understands, since many popular websites offer their content (or at least their interfaces) in different languages. Almost all clients send the `User-Agent` string, which should serve to identify the client software (and its version) for statistical purposes. Yet this is also being increasingly used to serve different versions of the content depending on the browser vendor, to work around specific bugs or take advantage of specific features, or to serve a mobile version of the page in case a mobile browser is identified. Finally, most clients send the `Referer` header, i.e., the URL of the previous web page from which a link to the currently requested object was followed: content can differ for different referrers. Given that none of these headers deterministically imply different content, their usage is non trivial, and we disregard them in the following.



(a) Cacheability, cache at OLT only.

(b) Traffic reduction, cache at OLT only.

Figure 4.10: Time evolution of daily statistics. Comparison of cacheability and traffic reduction calculated using three increasingly complex methods.

4.4.2 Object identification strategies

Object identification strategies leverage protocol information described in the previous section. Since many designs are possible, we select three representative strategies, that span the whole spectrum in terms of implementation complexity:

1. A simple strategy that only considers the URL and the ETAG.
2. A slightly more complex strategy that combines the ID from the previous point with the sizes of the objects.

3. A considerably more complex strategy that combines the ID from the previous strategy with an accurate tracking of the requested ranges.

The first two methods are rather simple and fast to compute (per-flow log, processing in the order of minutes for our datasets) whereas the third method comports a considerable usage of resources (per-range request log, processing in the order of hours). The first strategy is obviously the easiest to implement, as only the URL and (if present) ETAG are needed. The simple model behind it is a "normal" cache that will fetch the whole object as soon as any part of it is requested.

The second strategy also considers the size of the objects. The total size of the object is extracted, in order of preference, from the `Content-Range` header, from the `Content-Length` header or finally from the real number of bytes transferred. This allows to distinguish with more accuracy when the same URL yields different objects, as their size will likely be different, making therefore unnecessary to track the other HTTP headers described in the previous section. Effects of range requests are still ignored.

Finally, the third strategy adds complete state tracking of all the requested ranges. In this scenario a request is considered cacheable only if it overlaps at least partially with a previous request. Only the range of effectively transferred bytes is counted, so a resumed interrupted download is not considered as a hit, provided there was no overlapping with any other previous request for that object. Objects themselves are identified with the same system as the second strategy. This strategy considers an object cacheable w.r.t. traffic reduction if at least one of the partial requests was a hit.

Fig. 4.10 shows the comparison of the cacheability and traffic reduction using the three systems, on 1-day timeslots. It can be seen that there is very little difference between the first two methods, and the third method is also not too distant. The reason for the lower value of the traffic reduction for the third method is that, in case of transactions that are fragmented in multiple range requests, most of the fragments are big (because of the type of objects that are usually fetched with range requests, as previously explained), and most of those requests are cache misses. With the first two methods those requests are considered as hits, because they all involve the same object, and therefore all the traffic they generate is considered cacheable whereas with the third method, requests for non overlapping ranges are considered non cacheable. Fig.4.11 shows the distribution of the real size of the objects transferred with and without range requests. It can be observed that not only the single transactions, but also the objects themselves that are transferred using range requests are larger than the ones that are transferred without range requests.

Observation 4.4.1. *Accurate object identification would benefit from a significant level of protocol details to be taken into account. Simpler object identification techniques, such as the URL+ETAG method lead to slight overestimation of traffic cacheability and reduction, but are comparatively extremely simpler to implement and thus justifiable as a good compromise for large-scale online traffic analysis.*

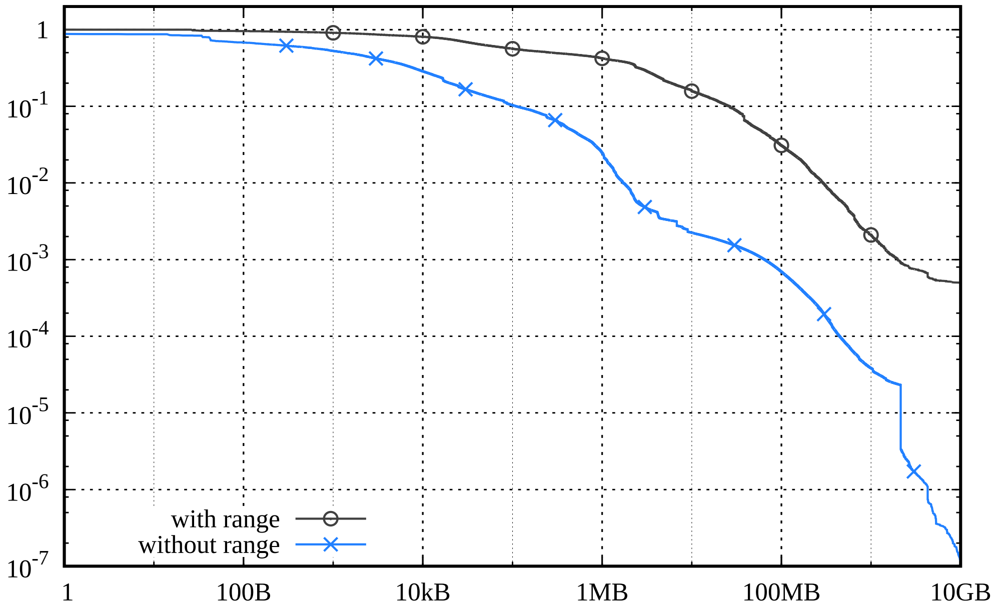


Figure 4.11: Distribution of the sizes of the objects requested with or without range requests. Reverse CDF.

4.5 Discussion

The significant potential gains we have measured in today's traffic do not apply to encrypted web applications (15% in the first dataset, 30% in the second, steadily increasing) which cannot be transparently cached. Caching TLS encrypted traffic is however going to be a significant issue for ISPs as it would require some form of interaction with the content provider, or the CDN on its behalf. Transparent caching of encrypted traffic might also be achieved by transparent interception of TLS tunnels, which implies some form of increased vulnerability at the user's client. We suggest instead to use CCN as the best fit technology to address all the drawbacks of currently available workarounds and providing caching as a transparent network primitive. We additionally assess technical feasibility in nowadays hardware, and argue that such a small amount of additional memory can be supported at high speed on current technologies. We finally identify steps for micro-CDN implementation, outlining arguments to support a CCN-based deployment as basic building block. Our analysis suggests that CCN-based solutions are close enough to be deployed in real ISP networks – its realisation is part of our ongoing work.

4.5.1 Technical feasibility

Previous sections have shown that significant benefits in terms of traffic reduction can be achieved at the cost of very limited additional memory. Indeed, a single ONT installed in a user's home would only need approximately 100MB of additional memory. Such memory, currently not available in optical devices, is available in the home gateways, that are equipped with enough CPU resources to easily manage 100MB of RAM at 1Gbps. Upstream to the OLT, 100GB memory in a router line card would be enough to provide the early shown gains and it seems feasible in current hardware [75]. Hence, the deployment of a micro CDN technology in the home would only require to implement the content-centric forwarding engine in the home gateway firmware – which again seems feasible due to the current development effort on several CCN/NDN prototypes.

4.6 Summary

In this thesis we provide evidence of the potential gains associated to the deployment of micro CDN technologies in ISP networks. Our analysis is grounded on a large dataset collected via the on-line monitoring of links between the access and the back-haul network of Orange France. Leveraging several months of continuous traffic monitoring in different vantage points in the operational network, we are able to support our design by an accurate characterisation of content dynamics.

The large data set we have used allowed fine-grained statistical analysis of content popularity dynamics, whose value goes beyond the primal objective of this thesis. Indeed, the analysis demonstrates the inadequacy of traditional models, like simplistic Zipf workloads, and their failure for prediction and dimensioning. The gains are striking: with a negligible amount of memory of 100MB on CPEs, the load in the access network (GPON) can be reduced by 25%, while embedding a 100GB of dynamic memory in edge IP router line cards can also reduce back-haul links load of about 35%.

Chapter 5

Big Data approach to cacheability analysis

As shown in Chapter 3, we had to handle pretty big datasets; the third dataset, in particular, proved to be impossible to handle on our server, therefore we decided to use a Hadoop cluster to perform the computations. Having so much computing power available enabled us to perform more accurate types of analysis, in particular regarding object identification.

This chapter will initially present an improved method to more accurately identify objects, in particular in presence of partial requests. A parallel approach to statistics computations is then presented, including a performance comparison with classical sequential implementations. Finally the results of the computation are presented for the largest datasets, including a performance analysis of different optimisations.

The statistics considered are Cacheability ($1 - \frac{N_o}{N_r}$), Traffic Reduction ($1 - \frac{R_u}{R}$), and Virtual Cache Size ($\sum_{o \in cacheable} V_o$), as introduced in 4.1.

5.1 Analytics

At a finer grain, several implementations of the cacheability, traffic reduction and virtual cache size analytics are possible: ultimately, all these indices depend on the way in which objects are identified, and the way in which their volume is computed.

5.1.1 Improving object identification

As illustrated in section 4.4, HTTP object identification can leverage different information such as e.g. object ID, range information, ETAGs, cookies, etc. The *object ID* is a 64-bit value calculated as the *FNV1a* hash of the URL of the object concatenated to its ETAG (if present). Similarly, volume estimation can leverage information such as advertised size (HTTP header information), or take into account effective size (HTTP payload actually transferred over the TCP connection). For instance, the effective size is less than the advertised one if the download is interrupted. In case of range requests, the advertised size is the size of the part of the object that is returned, and not the full size of the object, which is then reported in the range information.

Object identification and size estimation strategies leverage protocol information described above. We select the three strategies introduced in section 4.4, that span the whole spectrum in terms of implementation complexity:

— *ObjectID:* The simplest strategy, which only considers the Object ID (URL and the ETAG).

— *Size:* The slightly more complex strategy, which combines the ID from the previous point with the full sizes of the objects (advertised size or full object size in case of range requests).

— *Ranges:* The considerably more complex strategy, which combines the ID from the previous strategy with an accurate tracking of the requested ranges.

Accuracy vs Complexity Tradeoff

Roughly speaking, we face the following tradeoffs: the *first two methods may lead to overestimation of caching benefits, but they are rather simple and fast to compute.* These methods perform only per-request operations, require only a small amount of memory, are easy to implement using standard UNIX command-line tools and their processing time is in the order of minutes for the typical day in our dataset. The *third method is instead more accurate, at the price of a considerable usage of resources.* The latter method indeed performs per-range request operations, which both increase the required amount of memory as well as the processing time, and is not trivially implemented using only UNIX command line tools. We summarise the properties of these strategies in Tab.5.1, which reports rough projection of time and memory complexity based on our experience on smaller datasets[79]. For the sake of illustration, Fig.5.1 additionally quantifies cacheability and traffic reduction statistics computed from our previous small-scale dataset[79] with classical sequential algorithms.

More in detail, and as already introduced in section 4.4, the first strategy is obviously the easiest to implement, as only the URL and (if present) ETAG are needed. The log from the probe already contains this information in usable form (Object ID), so

Table 5.1: Comparison of the expected performance of the different object identification methods for our dataset

Method	**Accuracy** (Fig.5.1)		**Complexity** (Sec.5.3)	
(Sec.5.1 and 5.2)	**Object-wise**	**Byte-wise**	**CPU Time**	**Memory**
ObjectID	Fair	Fair	Minutes	Few GB
Size	Good	Fair	Minutes	Few GB
Ranges	Very good	Very good	Hours	Hundreds of GB

no additional processing is needed. However this possibly overestimates caching gains, which is not desirable.

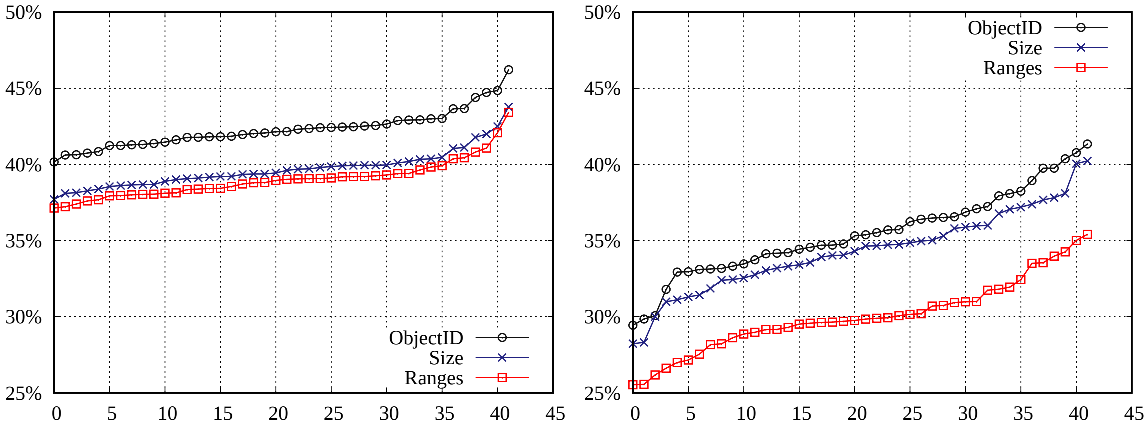
The second strategy also considers the size of the objects. The total size of the object is extracted, in order of preference, from the `Content-Range` header, from the `Content-Length` header or finally from the real number of bytes transferred. This allows to distinguish with more accuracy when the same URL yields different objects, as their size will likely be different, however effects of range requests are still ignored.

Finally, the third strategy adds a complete state tracking of all the requested ranges. In this scenario objects are identified as in the previous strategy, however a request is considered cacheable only if it overlaps at least partially with a previous request. Only the range of effectively transferred bytes is counted, so a resumed interrupted download is not considered as a hit, provided there was no overlap with any other previous request for that object. This strategy considers an object cacheable w.r.t. traffic reduction if at least one of the partial requests was a hit.

Observation 5.1.1. *In terms of accuracy, the left hand size of Fig.5.1 clearly shows that ObjectID is excessively simplistic in estimating object-level cacheability, whereas Size and Ranges provide very close results. At the same time, the right hand size of Fig.5.1 shows that both ObjectID and Size are excessively optimistic in estimating byte-level traffic reduction, unlike Ranges. It follows that this latter strategy is the most useful for our purposes.*

Implementation considerations

Some additional implementation details are worth highlighting, even in the relatively simple sequential case. Notice that instead of using a comfortable scripting environment, to optimise for speed and memory complexity we opted for a lower-level implementation in C. It follows that our sequential implementation is not very flexible, but instead highly optimised.



(a) Cacheability

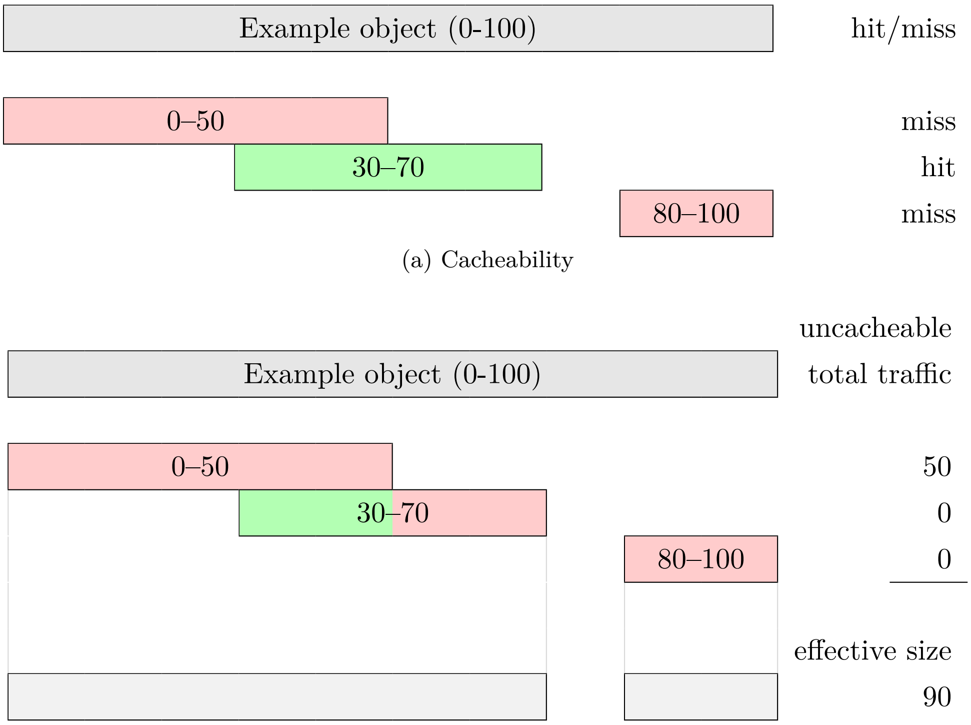
(b) Traffic reduction

Figure 5.1: Comparison of the accuracy of the different object identification methods for the [79] dataset. The Y axis reports cacheability and traffic reduction analytics, computed over daily intervals, which are then ranked from lowest to highest gain on the x-axis.

Complexity of the first two strategies (i.e. *ObjectID* vs *Size*) is basically the same. These strategies differ slightly in the way objects are identified: both methods keep the *object ID* and the *effective size* of the object (as the maximum observed value for that object); the second method additionally considers the *advertised size* to further distinguish between distinct objects with the same ID (but different advertised size).

A running sum is kept for the three analytics of interest. For cacheability, it is required to just keep track of the *number of total requests* N_r and the number of *unique objects* N_o: the former is updated at any new requests (i.e. HTTP GET), whereas the second is simply incremented whenever a new object is added to the hashtable. Estimating traffic reduction requires to keep track of *the total traffic generated* R and the *amount of cacheable traffic* R_c: the former is update at any new requests, whereas the cacheable traffic is computed as the sum of all traffic generated by objects seen more than once, whose count is tracked in the hashtable (and updated at each requests provided that the count equals or exceeds 2). For virtual cache size, we keep a running sum of the *effective size* V_o of the objects that are seen more than once, but obviously only once per object: also this last statistic uses the object request count tracked in the hashtable (but is updated only when the count equals 2).

As for *Size*, objects are identified with object ID and advertised size, and the effective size is calculated as the number of unique bytes seen. Similarly to the previous cases, analytics are tracked with running sums. For cacheability, we keep a running sum of the



(a) Cacheability

(b) Traffic reduction

Figure 5.2: Example of hit and miss in presence of range requests and partial downloads. Cache hits are highlighted in green, misses are highlighted in red. Since there is a hit and given how traffic reduction is calculated, all the traffic generated by the object is considered cacheable.

hits and the total number of requests. For traffic reduction, we keep a running sum of the effective traffic for cacheable objects (that is, objects that have at least 1 hit). For the virtual cache size, we keep a running sum of the effective size of all the cacheable objects.

Ranges is more complex: for each object, it needs to keep track not only of its ID, its advertised size and effective size, but also of the *number of hits*, and especially *the list of requested ranges*. The number of hits is necessary because, accounting for range requests, it is possible to see an object request more than once without having a hit, e.g. in case non-overlapping parts of the object are requested, like for chunked or resumed downloads. A request is only considered a miss if no bytes of the requested range of the requested object were requested previously, otherwise it is considered a hit. Partial downloads are also considered as ranges for the observed intervals, and partial downloads of range requests are also properly accounted for.

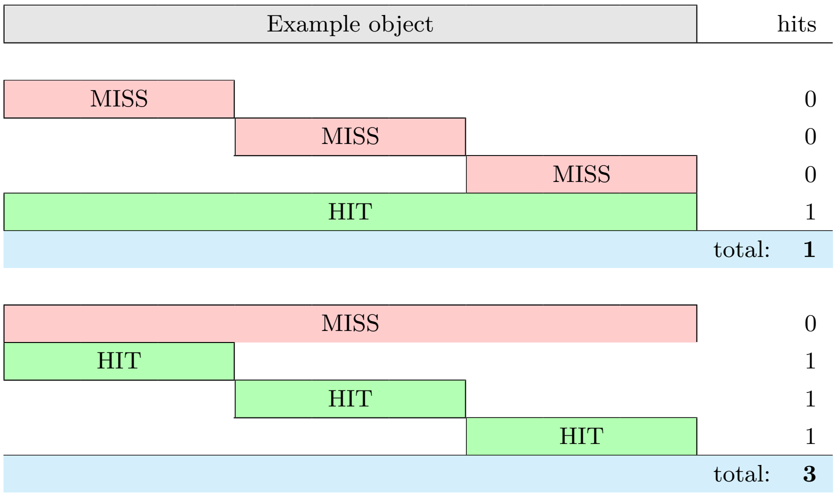


Figure 5.3: Visualisation of the importance of the arrival order for counting hits when considering partial requests.

Notice further that the order in which requests arrive is important: for example, an object that is first requested in two chunks and then is requested again entirely will generate only one hit, while the same requests in a different order will generate two hits. This is illustrated in Fig. 5.2 and Fig. 5.3, which show an example of possible criteria for considering a request as a hit or as a miss.

However, differently from the previous cases, range requests make the associate state more complex, and object tracking more involved. More precisely, as shown in Fig.5.2a, any request that overlaps even partially with any previous requests is considered a hit. Counting the number of hits therefore involves *comparing each partial request with all the previous ones* to check if there is an overlap.

Interval merging, is instead needed to properly calculate the virtual cache size. Interval merging is shown in Fig.5.2b, and it is actually performed step by step as new requests arrive, rather than at the end. The merged intervals are also used to check and count the hits, as discussed previously. Keeping only the merged intervals saves time during the computations, since each new interval has to be checked against a smaller number of other intervals.

5.1.2 Input dataset

While it is outside of the scope of this thesis to describe HACkSAw[80], the software tool that we deployed in a passive probe in the operational network of Orange, we however need to briefly comment on some details of the dataset. While we deployed HACkSAw in different levels of aggregation, in this work we focus on monitoring 2x10Gbps links at a PoP serving a population of about 40,000 users. The software outputs a plaintext database, with about 25 columns per file, and one file for (i) TCP flows, (ii) UDP flows, (iii) HTTP transactions, (iv) DNS requests. Nearly 200,000,000 rows per day are generated, which amounts to about 100GB daily.

Statistics about the total and daily volumes are tabulated in Tab.3.2, which also reports a comparison with datasets used in our previous work [79] and in related work. It can be seen that the dataset we considered here has the *longest duration* (3× our previous work and over 10× longer than in the related literature), has the *largest number of requests* (70× our previous work and over 4× the largest dataset in related literature), the *largest number of objects* (50× our previous work and about 3× the largest dataset in related literature), that correspond to the *largest traffic volume* (100× our previous work and about 10× the largest dataset in related literature).

The sheer dataset size poses significant computational challenges, even for our highly optimised ANSI C implementation: it can be seen that the big dataset has more than $8{\cdot}10^9$ distinct objects, with a daily average of $90{\cdot}10^6$. Grouping by sorting is prohibitively time consuming, and needs too much temporary storage. We also project that computing the previously illustrated analytics grouping by day using a hashtable is also prohibitive. We lower bound the memory size of such an hashtable by considering that, at a minimum for each object we need at least to keep track of (i) object ID, (ii) advertised size, (iii) effective size, (iv) timestamp, and (v) number of hits. Additionally, each object needs (vi) a pointer to the next object in the hashtable, (vii) a pointer to the list of chunks, and (viii) at least one chunk with pointer to next, (iv) a start and (x) end offsets. Each of those items takes up 8 bytes for a total of at least 80 bytes per object. Considering the rate of distinct objects per day $\Lambda = 9 \cdot 10^7$ objects/day, the size $S = 80$ bytes/object of each object, and the length $t = 132$ days of out dataset, we need $S\Lambda t = 80{\cdot}(9{\cdot}10^7){\cdot}132 \approx$ 950GB of memory, just to compute the three key statistics mentioned above, across the overall dataset. This is of course a very optimistic lower bound since we are overlooking the fact that some objects will have more than one chunk. On top of that, we are also ignoring all the other data structures used to implement the hashtable.

Of course, according to the above projections we estimate that our ANSI C implementation should still be capable to compute daily statistics on the new dataset (where some 10GB of RAM should suffice). At the same time, it also appears that big-data approaches such as Hadoop seem an interesting option to scale the analysis of the current dataset, with the added benefit of gaining horizontal scalability to further scale this up analysis in time (e.g., extended duration), space (e.g., considering multiple probes, or a

```
A0 = LOAD '$inputfiles' using PigStorage('\t') AS (f1:chararray, f2:chararray,
 f3:chararray, f4:chararray,f5:chararray, f6:chararray, f7:chararray, f8:chararray,
 f9:chararray, f10:long,f11:chararray, f12:chararray, f13:chararray, f14:chararray,
 f15:chararray, f16:chararray, f17:chararray, f18:long);

in = FOREACH A0 GENERATE f12 AS ID, f18 AS Traffic,
 (f10/(3600L*1000000L)+2L)/24L AS TimeStamp;

o = foreach (group in by (ID,TimeStamp)) generate FLATTEN(group) as (id,ts),
    (long)COUNT(in) as nreq:long, (long)SUM(in.Traffic) as tr:long,
    (long)MAX(in.Traffic) as size:long;

ct = foreach (group o by ts) {
 ids = o.id;
 objects = DISTINCT ids;
 cac = filter o by (long)nreq>1L;
 generate (long)(group*24L-2L)*3600L as ts:long, (long)COUNT(objects) as no:long,
  (long)SUM(o.nreq) as nreq:long, (long)SUM(o.tr) as tr:long,
  SUM(cac.tr) as cact:long, SUM(cac.size) as vc:long;
}

STORE ct INTO '/User/pigoutput' using PigStorage(' ');
```

Figure 5.4: PIG code of ObjectID. While not particularly elegant, the code is however very compact with respect to the about 1,200 lines of code our ANSI C implementation.

larger user-base) or line-rate (e.g., moving the probe to a higher level of aggregation),

5.2 Parallelising caching analytics

We parallelise the analysis of caching analytics by leveraging a big-data cluster in Orange. While for reason of confidentiality we cannot fully disclose details of the cluster, for the purpose of this work it is largely sufficient to say that the cluster is based on Apache Hadoop and it has over 2,000 cores.

The workflow has been automated to gather up-to-date statistics without requiring human intervention. A script is run periodically on the probe to compress the logs of the previous hours, which are then transferred to a gateway that will decompress and inject the data in the Hadoop cluster. The compression is needed to minimise the bandwidth requirement between the probe and the cluster, which are in different networks.

Oozie is used both to automate and coordinate the compute jobs. The Oozie job serialises the first and second map-reduce jobs, starting the second one only if the first one was successful. The Oozie job runs the task on the cluster at regular intervals (namely: hourly and daily) and the results are then copied to a web server for display.

In the reminder of this work, we limit ourselves to analyse and benchmark the analytics, and disregard all aspects such as automation that, albeit extremely important from an operational viewpoint, are however less relevant from a scientific standpoint.

5.2.1 Design choices

There are several approaches that can be used to process large amounts of data. Traditional SQL databases, for example, are very well suited to process large amounts of data. We however need not only to analyse a sheer amount of data, but also to perform fairly complex calculations on the data (e.g. range reconstruction/consolidation). At the same time, while traditional SQL databases are not suitable for our purposes, there exists SQL-like interfaces built on top of the Hadoop framework, such as PIG or Hive, that are thus worth considering.

For illustrational purposes, Fig.5.4 shows the PIG code of ObjectID strategy for object identification. Gains are clear by considering that our ANSI C implementation amounts to over 1,200 lines of code, which is in part due to the lack of even basic structures such as hashtables in ANSI C.

However, simplicity is not without cost. Indeed, PIG hides the complexity of a native implementation in Map-Reduce, but at the same time does not allow to optimise the usage of Map-Reduce jobs as a native Java implementation would allow. In the rest of this section, we describe our native Java implementations of the early illustrated cacheability analytics.

5.2.2 Map-reduce analytics

We implement the cacheability, traffic reduction and virtual cache size analytics with a *two-stages* Map-Reduce configuration. Irrespectively of the specific object identification method, the general architecture of the map-reduce jobs is similar, and schematically illustrated in Fig.5.5.

The two stages differ slightly depending on the object identification method. In the *ObjectID* and *Size* cases, the *first map* parses the input lines and groups the requests by timeslot and ID (or extended ID, that is the object ID together with the advertised object size,in the *Size* case). The *first reduce* computes the aggregate per-object/per-timeslot values. The *second map* is just the identity function, while the *second reduce* step aggregates all the values for all the objects in each timeslot.

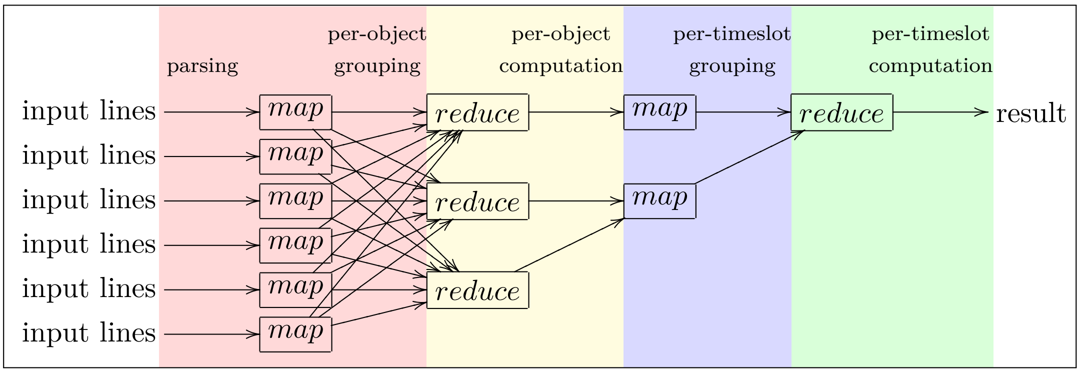


Figure 5.5: Dataflow in the two-stage Map-Reduce. Colours highlight the various stages.

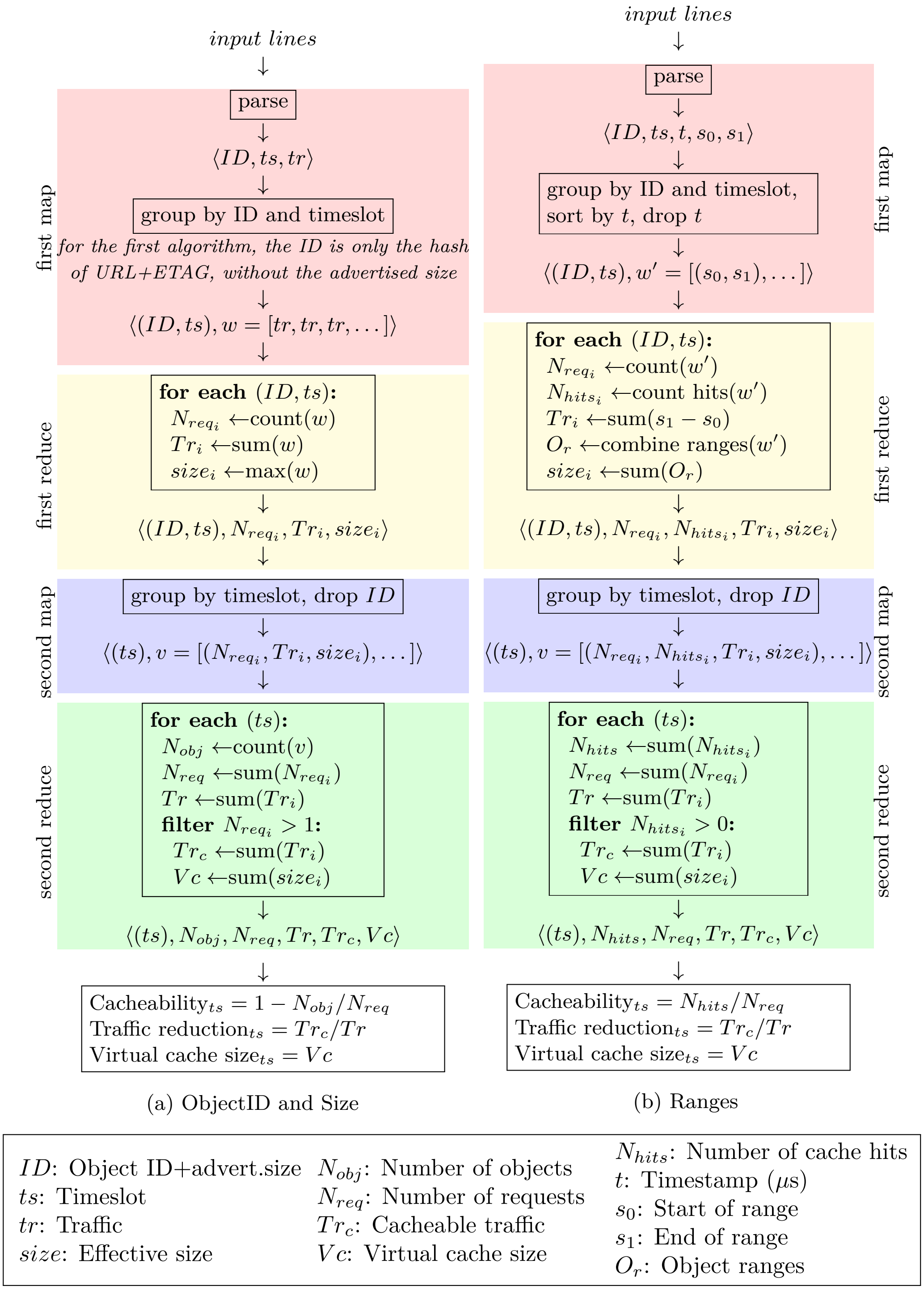
In the *Ranges* case, the order of arrival is especially relevant for accuracy, so that this algorithm needs to sort request by time inside each timeslot after the first map. Then, the first reduce step merges all requests for different chunks of the same object, properly calculating the number of hits and the amount of generated traffic. After the first reduce, for each object, we have the number of requests, number of hits, total traffic and number of unique bytes.

Analytics are presented in more details in Fig. 5.6, which graphically shows the dataflow in greater detail and the actions performed in each phase. The first step, common to all algorithm is input parsing. Text input lines are parsed, invalid lines are discarded; the number and type of fields parsed is instead algorithm dependent. Invalid lines can arise from bugs in the capture tool, data corruption on the path to the Hadoop storage or finally corruption in the Hadoop storage itself. We now comment each of theses phases, for all algorithms, in more depth.

ObjectID and Size analytics

Analytics based on *ObjectID* or *Size* object identification strategies are represented in the left hand side of Fig. 5.6. For each request, the first two algorithms (*ObjectID*, *Size*) parse the ID, the timeslot and the amount of traffic generated for that request. The ID is only the Object ID for the *ObjectID* algorithm and is the concatenation of Object ID and advertised size for the *Size* algorithm.

The requests are then grouped by ID and timeslot. Each slot will therefore contain all the requests for a specific object in a specific timestlot, with ID and timeslot as key, and the list of generated traffic as value. For each slot three statistics are calculated: the number of requests for that object in that timeslot, which is simply the length of the


input lines
parse
$\langle ID, ts, tr\rangle$
group by ID and timeslot
for the first algorithm, the ID is only the hash of URL+ETAG, without the advertised size
$\langle (ID, ts), w = [tr, tr, tr, \dots]\rangle$
first map
for each (ID, ts):
$N_{req_i} \leftarrow \text{count}(w)$
$Tr_i \leftarrow \text{sum}(w)$
$size_i \leftarrow \max(w)$
$\langle (ID, ts), N_{req_i}, Tr_i, size_i\rangle$
first reduce
group by timeslot, drop ID
$\langle (ts), v = [(N_{req_i}, Tr_i, size_i), \dots]\rangle$
second map
for each (ts):
$N_{obj} \leftarrow \text{count}(v)$
$N_{req} \leftarrow \text{sum}(N_{req_i})$
$Tr \leftarrow \text{sum}(Tr_i)$
filter $N_{req_i} > 1$:
$Tr_c \leftarrow \text{sum}(Tr_i)$
$Vc \leftarrow \text{sum}(size_i)$
$\langle (ts), N_{obj}, N_{req}, Tr, Tr_c, Vc\rangle$
second reduce
Cacheability$_{ts} = 1 - N_{obj}/N_{req}$
Traffic reduction$_{ts} = Tr_c/Tr$
Virtual cache size$_{ts} = Vc$
input lines
parse
$\langle ID, ts, t, s_0, s_1\rangle$
group by ID and timeslot, sort by t, drop t
$\langle (ID, ts), w' = [(s_0, s_1), \dots]\rangle$
first map
for each (ID, ts):
$N_{req_i} \leftarrow \text{count}(w')$
$N_{hits_i} \leftarrow \text{count hits}(w')$
$Tr_i \leftarrow \text{sum}(s_1 - s_0)$
$O_r \leftarrow \text{combine ranges}(w')$
$size_i \leftarrow \text{sum}(O_r)$
$\langle (ID, ts), N_{req_i}, N_{hits_i}, Tr_i, size_i\rangle$
first reduce
group by timeslot, drop ID
$\langle (ts), v = [(N_{req_i}, N_{hits_i}, Tr_i, size_i), \dots]\rangle$
second map
for each (ts):
$N_{hits} \leftarrow \text{sum}(N_{hits_i})$
$N_{req} \leftarrow \text{sum}(N_{req_i})$
$Tr \leftarrow \text{sum}(Tr_i)$
filter $N_{hits_i} > 0$:
$Tr_c \leftarrow \text{sum}(Tr_i)$
$Vc \leftarrow \text{sum}(size_i)$
$\langle (ts), N_{hits}, N_{req}, Tr, Tr_c, Vc\rangle$
second reduce
Cacheability$_{ts} = N_{hits}/N_{req}$
Traffic reduction$_{ts} = Tr_c/Tr$
Virtual cache size$_{ts} = Vc$
ID: Object ID+advert.size
ts: Timeslot
tr: Traffic
$size$: Effective size
N_{obj}: Number of objects
N_{req}: Number of requests
Tr_c: Cacheable traffic
Vc: Virtual cache size
N_{hits}: Number of cache hits
t: Timestamp (μs)
s_0: Start of range
s_1: End of range
O_r: Object ranges

(a) ObjectID and Size

(b) Ranges

Figure 5.6: The three algorithms. The same colours as in Fig.5.5 are used to help identify the map-reduce stages.

list; the total traffic generated by that object, calculated as the sum of the list; and the effective size of the object, calculated as the maximum value in the list.

The elements are now grouped by timeslot only: the timeslot is thus the key and the value is a list of tuples, one for each object in the timeslot, with the values calculated in the previous step. The final statistics are now calculated. The number of distinct objects is calculated simply as the length of the list. The total number of requests is calculated as the sum of the requests for each object; the total traffic is calculated as the sum of the traffic generated by each object, the cacheable traffic is calculated as the sum of the traffic generated by the objects that were requested more than once; the virtual cache size is calculated as the sum of the effective size of the objects requested more than once.

Finally the desired statistics for cacheability and traffic reduction are calculated as $1 - N_{obj}/N_{req}$ and Tr_c/Tr respectively. The virtual cache size is simply the value calculated in the previous step.

Ranges analytics

In the case of *Ranges*-based object identification, For each request the algorithm parses the request ID (as in *Size*, the ID is the concatenation of Object ID and advertised size), the timeslot, timestamp, start of the request (which is normally the first byte, but can be different in case of range requests), and end of the request (calculated as the sum of the start offset plus the effective size).

The requests are then grouped by ID and timeslot, and sorted by timestamp. After sorting, the timestamp is not needed any longer and is thus dropped. Sorting is important to accurately calculate the cache hits. Each slot will therefore contain all the requests for a specific object in a specific timeslot, with ID and timeslot as key, and the list of pairs $(start, end)$ as value.

For each slot, several statistics are calculated: the number of requests for that object in that timeslot, which is simply the length of the list; the total traffic generated by that object, calculated as the sum of the length of each request; the number of hits, calculated as the number of requests for the object that overlap at least partially with previous requests for the same object; and the size, calculated as the number of unique distinct bytes.

The elements are now grouped by timeslot only: the timeslot is once again the key, and the value is a list of tuples, one for each object in the timeslot, with the values calculated in the previous step. The final statistics are now calculated, similarly to the other algorithms. The number of hits, the number of requests and the total traffic are all calculated by summing the respective per-object values. Cacheable traffic and virtual cache size are calculated as the sum respectively of traffic and size of the objects with at least one hit.

Finally the desired statistics for cacheability and traffic reduction are calculated as $1 - N_{hit}/N_{req}$ and Tr_c/Tr respectively. The virtual cache size is simply the value calculated in the previous step.

5.3 Results

In this section we report benchmarks of the analytics, mostly focusing on computational and memory complexity. We first illustrate the coding simplicity vs computational overhead tradeoff by comparing ANSI C, PIG and Map-Reduce implementations. We then optimise the running time of the most accurate and complex algorithm (Ranges), using compression and binary comparators. We finally illustrate results of the analytics.

5.3.1 Coding Simplicity vs Computational Overhead Tradeoff

As stated earlier, a natural tradeoff arises between the amount of control on a specific tool and the expected level of performance optimisation that the tool allows. On the one extreme, we have total control over our low-level ANSI C implementation: while in this case it is easy to optimise for memory, and for performance on a single core, however we are limited to serial operations and face a computational complexity bottleneck. On the other extreme, PIG allows for very simple declarative-like queries over large datasets that benefit from parallel execution. This allows to express some of our analytics very compactly, with however a loss of control on the execution workflow, that is delegated to the PIG interpreter. In the middle, the Java Hadoop implementation retains a certain amount of control, and hence an expectedly lower overhead (provided that the analytics design is sound), with the added benefits of parallel execution. In this execution, we employ 255 mappers and 100 reducers.

The above tradeoff appears very crisply in Tab. 5.2, which shows a comparison of computational and memory complexity of serial (ANSI C) vs parallel (PIG and Java Map-Reduce) implementations of our analytics. Specifically, in the serial case we contrast the simplest vs the most complex algorithm, while we only consider the simple algorithm for the parallel case, for now. The table reports both the absolute performance, as well as the relative performance with respect to the ANSI C implementation of *ObjectID*

Table 5.2: Comparison of computational and memory complexity of serial (ANSI C) vs parallel (PIG and Java Map-Reduce) implementations of our analytics. Ratios in brackets are relative to the ANSI C implementation of *ObjectID* used as a reference: thus, ratios smaller (larger) than one correspond to performance gain (loss).

	Serial implementation		**Parallel implementation**	
Method	ObjectID	Ranges	ObjectID	ObjectID
Language	ANSI C	ANSI C	PIG	Java Map-Reduce
Runtime	25min	62min	38min	4min
	–	(1.8×)	(1.5×)	(0.16×)
Memory	8GB	12GB	n.a.	300GB
	–	(1.5×)	–	(37×)

used as a reference: thus, ratios smaller (larger) than one correspond to performance gain (loss). Notice that the serial vs parallel ratios are to be interpreted as *qualitative* as opposite to quantitative comparison: indeed, execution happens on hardware with similar, albeit not identical configuration.

At the same time, this qualitative comparison allows to gather very useful lessons. First, notice that despite parallel execution, the overhead of *ObjectID* in PIG appears to be so large that the actual computation of the results takes longer [1] in the cluster than on a single-core serial execution. It follows that an implementation of more complex algorithms in PIG does not seem to be worth pursuing further for our analytics.

Second, notice that Map-Reduce implementation largely benefits of parallel execution, with running time that reduces by almost one order of magnitude (0.16×) with respect to serial execution in the simplest object identification case. At the same time, notice that the Map-Reduce framework does consume a considerable amount of memory: while this is not a critical issue (since, provided that there is enough memory, the jobs complete faster), it could be potentially an issue depending on the load of the cluster (i.e. the amount of jobs and their memory requirements as well). It follows that we expect Map-Reduce to be worthwhile investigating also for more complex algorithms.

5.3.2 Illustration of Map-Reduce Workflow

We now closely look at the execution pattern of our Map-Reduce implementation. For the sake of illustration, Fig.5.7 reports a waterfall diagram of the execution time of

[1] Note that we do not report memory consumption as this is not clearly available with our account rights in the cluster

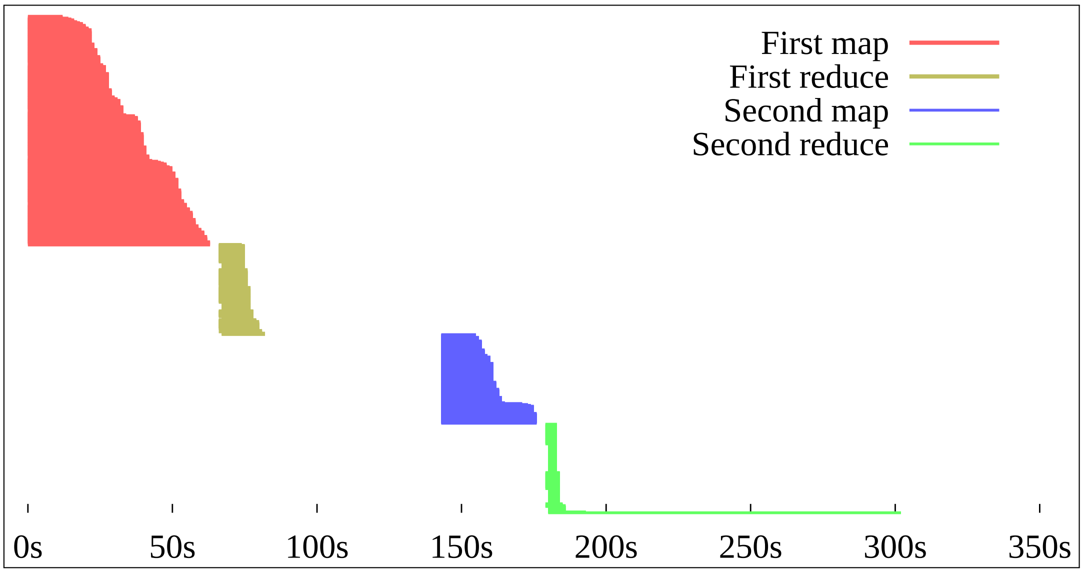


Figure 5.7: Waterfall diagram of a run of the Map-Reduce jobs for *Ranges*

one run of the most complex analytics, i.e. *Ranges*. Map-Reduce phases in the waterfall use colour codes that are consistent with those used in the previous section.

Several takeaways can be summarised from the picture. First, notice that the *overall execution time* of the complete workflow remains of the same order of magnitude that those shown in Tab.5.2 for the simplest algorithm (*ObjectID*). This is striking, especially since our optimised ANSI C implementation suffered a runtime degradation of about a factor of two: implicitly, this testifies the existence of a *non-negligible overhead* in the Map-Reduce system that not only affects the memory complexity (as per Tab.5.2), but also the computational complexity. At the same time, this overhead is only apparent for relatively simple tasks, but it gets diluted for more complex operation (at least for our analytics). This is especially reassuring, as the runtime reduction (With respect to *ObjectID* ANSI C implementation) stays at about 0.16× in the Map-Reduce implementation for both *ObjectID* as well as *Ranges* (i.e., more complex operations do not cause higher overhead).

Second, consider the *footprint and duration of each Map-Reduce phase.* In terms of footprint, the number of job in the first phase is correlated with the mappers (255) and reducers (100). Since the mapping in the second phase is the identity function, it follows that the number of parallel executed jobs remains 100 in the second phase. In terms of duration, notice that there is a non marginal variability of job duration in the first map phase (red colour), with execution time of individual jobs distributed in the

[10,60]sec range. The first reduce phase (yellow colour) starts immediately after the first map ends, and individual jobs complete faster, yet writing to disk delays the start of the second phase. This can be seen as a pause happening between the [70,140]sec range.

Job footprint in the second map phase (blue colour) are, as we commented, limited to 100, with duration that span in a [10,20]sec range in most of the cases, with a significant fraction however finishing after about 30sec. Finally, the second reduce phase starts (green colour); in this phase, there is actually a single reducer doing actual computation to consolidate all records in overall cacheability analytics over all clients and timeslots. It follows that 99 out of 100 jobs complete in less than 5 seconds, whereas the 100th job lasts for about 120sec. Overall, the duration of this specific execution for *Ranges* is 300sec, very much in line with simpler object identification techniques of *ObjectID*, but with the added benefit of increased accuracy.

5.3.3 Optimising Map-Reduce Running Time

While these observation are insightful about the general pattern, they are not statistically representative of the expected running time. To optimise our workflow in a statistically significant results, we proceed as follows. We first let the system collect a large enough dataset (30 days), and then process this dataset (2 runs for each day) under different system-level optimisation. By optimisation we essentially mean two technical solutions: (i) implementation of a binary comparator and (ii) use of compression to store intermediate values. An important and aspect worth pointing out is that our analytics are run in the production cluster and have an operational value, we preferred to run batches of tests during the month of August 2015, when the amount of traffic is lower than the rest of the year: as traffic level is lower than average, caching is thus less interesting to absorb excess traffic and we can afford losing one month worth of data.

As for (i), recall that the sorting is performed on the keys, which are objects. Normally the Hadoop system would deserialise the incoming objects to compare them, which is a time consuming activity. For this reason we decided to write a raw comparator to compare the serialised keys, thus skipping the costly Java object deserialisation step. This means that the raw comparator must parse the bitstream of the serialised keys, which is not a trivial task, but it brings speed advantages.

As for (ii), given the amount of intermediate data that needs to be written to disk between the two map-reduce jobs, compression may seem an interesting option: by compressing the data, the amount of data that has to be written to disk reduces (i.e. the gap between the first and second phase), and the writing time to disk may decrease

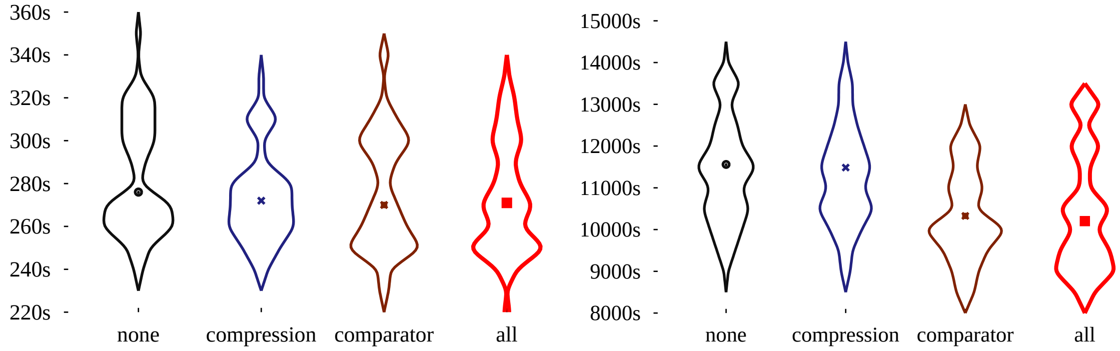
Table 5.3: Average Map-Reduce statistics for daily dataset processing with Ranges identification, and several optimisations (compression, binary comparator and combination thereof)

	Optimisation			
	None	**Compression**	**Comparator**	**Both**
HDFS disk read	66.5GB	65.6GB	66.5GB	65.6GB
HDFS disk written	1.5GB	630MB	1.5GB	630MB
Map tasks	255	255	255	255
Reduce tasks	100	100	100	100
Map input records	$250 \cdot 10^6$	$250 \cdot 10^6$	$250 \cdot 10^6$	$250 \cdot 10^6$
Cumulative CPU time	154 min (2.57 h)	165 min (2.74 h)	147 min (2.46 h)	146 min (2.43 h)
RAM used	301GB	300GB	291GB	291GB
Real time	4.08 min	4.06 min.	3.95 min	3.81 min
Speedup factor	37.7	40.6	37.3	38.6

as well. At the same time, compression takes some amount of CPU time, hence it is not obvious to forecast the amount of gain in the overall execution time. Well aware of this tradeoff, we decide to compress the data using Snappy, a fast compression algorithm developed by Google. The reason we select Snappy, despite there being many other compression algorithms with better compression ratios, is precisely its light footprint: indeed, spending too much time compressing the data would ultimately counter the gains of having less data to write to storage. According to our preliminary tests, Snappy is even faster than LZO or Gzip with the fastest settings. On the other hand, it has a worse compression ratio, generally data compressed with Snappy is 20% to 100% bigger than with other algorithms. While compression speed is similar to LZO, decompression is a lot faster; compared to Gzip, instead, Snappy is about one order of magnitude faster. Finally, due to license issues, Snappy is easily found in most Hadoop deployments, whereas other libraries must always be installed manually.

Details of this benchmark are reported in Tab.5.3 and Fig.5.8. Specifically, the table reports the average values over 30x2 experiments per each configuration; the amount of resources are referred to process one timeslot of one day: it can be seen that, while the total computational time across all cores is very high (some hours), the real-time duration is very small (few minutes), therefore enabling almost interactive analysis of the traffic.

Fig.5.8 additionally reports details about the most interesting performance indicators, namely the duration of the whole process Fig.5.8-(a) as well as the cumulative CPU



(a) Real time

(b) CPU time

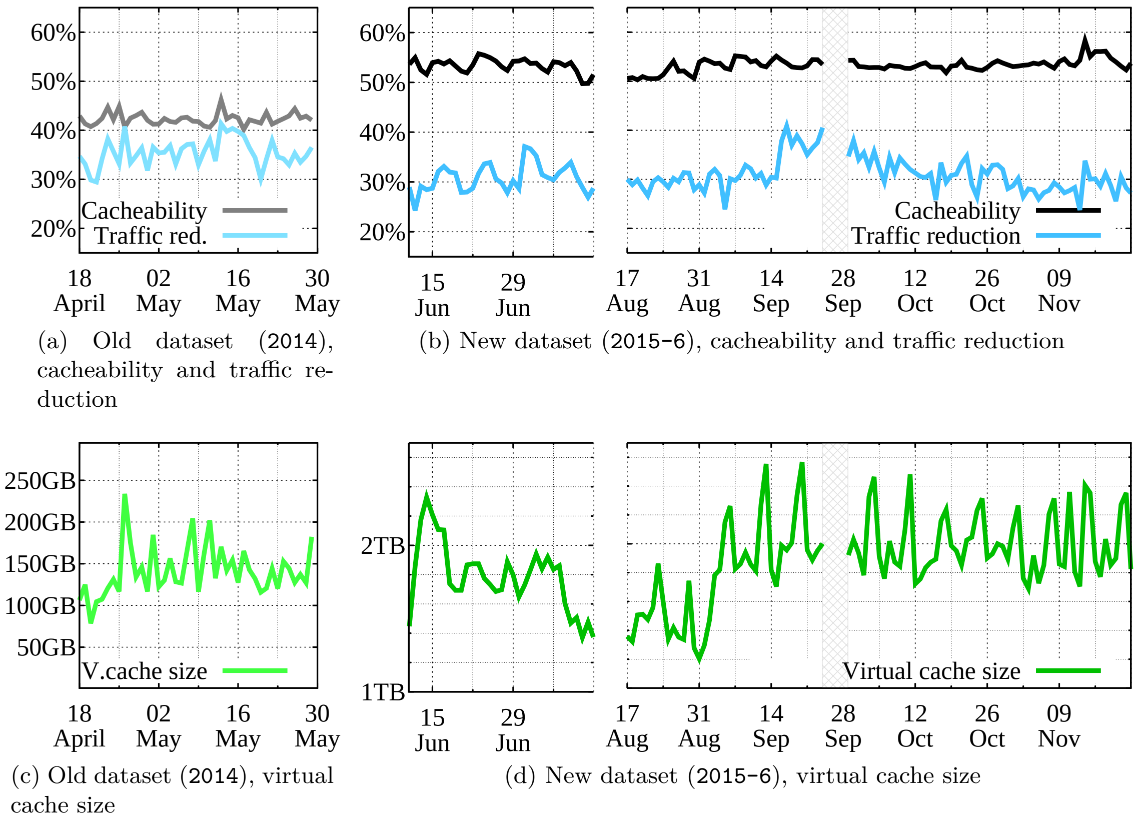
Figure 5.8: Violin plots of real time (a) vs cumulative CPU time over all cores (b) for the Ranges identification. Notice a consistent reduction in the cumulative CPU time (nearly 1000secs), to which however corresponds only a marginal gain when both optimisation are in use.

time over all jobs Fig.5.8-(b). The duration of the process is useful from the viewpoint of users of these analytics, while the cumulative CPU time is useful from the viewpoint of the cluster maintainer. For instance, we expect compression to potentially reduce the overall duration, at the price of an increase of the cumulative CPU time. Statistics in Fig.5.8 are reported as violin plots: the shape of the violin is the PDF of the statistics of interest, which is made symmetric in the visualisation (i.e., the area of each violin equals to twice the area under the PDF, hence 2 by definition). The violin also report the median reference as a point in the plot.

Enabling both (i)+(ii) optimisations, one can notice a consistent reduction in the cumulative CPU time (nearly 1000secs), to which however corresponds only a marginal real-time gain when both optimisation are in use. This is due to the fact that compression has a very limited impact on the runtime: the disk bottleneck is *reading* the input, the amount of intermediate *writing* saved by using compression is less than 3%, hence a marginal improvement. Using the binary comparator, instead, brings some clear advantage in the cumulative time: at the same time, the gain spread over all jobs reduces in the overall duration, where as we have seen there exist correlation between phases, so that reducing the average job duration helps only in part.

5.3.4 Analytics Output

Finally, we illustrate the output of the analytics gathered in the operational network with our automated workflow. While the analysis of these results is not the main topic



(a) Old dataset (2014), cacheability and traffic reduction

(b) New dataset (2015-6), cacheability and traffic reduction

(c) Old dataset (2014), virtual cache size

(d) New dataset (2015-6), virtual cache size

Figure 5.9: Illustration of the analytics on the [79] dataset (2×1Gbit/s, 42 days) along with the results calculated for the new dataset (2×10Gbit/s, 132 days; Note the two interruptions in the plot: the August interruption is a voluntary one to perform the benchmarks and the MapReduce workflow tuning, the October one is due to a small hardware issue on the probe.

of this work, we believe to be important to provide at least a brief comment. Indeed, the side effect of this work is precisely of being able to peek into larger datasets, so that comparing new vs established results[80] is what this work enables.

The plots in Fig.5.9 report execution of the serial analytics on the [79] dataset (2×1Gbit/s, 42 days) alongside with the results of the parallel MapReduce analytics computed over the the new dataset (2×10Gbit/s, 132 days). Top plot reports the cacheability (black line) and traffic reduction (blue line), while bottom plot reports the virtual cache size (green line). Note the two interruptions in the plot: the August interruption is a voluntary one to perform the benchmarks and the MapReduce workflow tuning, the October one is due to a small hardware issue on the probe. Notice that cacheability and traffic reduction remain into close ranges: cacheability jumps from slightly more than 40% in the 1Gbps dataset to slightly more than 50% in the 10Gbps dataset; similarly, traffic reduction decreases from slightly more than 30% to slightly less than 30%.

Conversely, the virtual cache size required to attain these caching performance significantly increases, by more than one order of magnitude, jumping from over 100GB to over 2TB. In this scenario, the ability to serve at 10Gbps line rate, with caches that are for reasons of costs cannot be entirely made of DRAM storage becomes of primary importance, reinforcing the need of designs such as [75].

5.4 Summary

In this chapter, we showed how to exploit a MapReduce cluster to continuously assess cacheability statistics gathered from passive analysis of traffic flowing on high-speed links in an operational network. Our main contributions are: (i) to motivate the use of a parallel and horizontally scalable workflow; (ii) to design parallel analytics in PIG and MapReduce; (iii) to experimentally compare serial vs parallel analytics, as well as PIG vs MapReduce; (iv) to perform a thorough benchmark and optimisation of our MapReduce workflow.

We have also shown that, by using Hadoop and a native Java MapReduce implementation, a speedup greater than a factor of six is achievable, in comparison with a serial ANSI C implementation. This achievement enables the analysis of datasets that are by far larger than what has been done in the community so far – where by far means over 10× in duration, 10× in the number of requests and 10× in the distinct objects with respect to the largest dataset analysed in the available literature. Given the horizontal scalability of Hadoop MapReduce, we expect these analytics to be future proof – i.e. scale further in space, time or line rate – at no additional cost.

Chapter 6

Conclusions and perspectives

In this chapter we summarise the goals achieved in this thesis, the implications they have for the current network operators, and we provide a perspective on possible future research.

6.1 Summary

First, in Chapter 3, we illustrated the methodology used to capture the traffic traces used in this thesis, including a detailed overview of the design and implementation of the high-performance software tool used. We also introduced the metrics used to measure the amount of traffic or requests that can be saved, the network topology where the probe running our tool was placed, and some details about the characteristics of the traces themselves.

Then in Chapter 4 we analysed the traffic traces. First we determined that a timescale of one day is ideal, and then used the traces to simulate three scenarios, with caches in the backhaul, at the customer's home router, or both. The metrics introduced previously give us the upper bound of about 40% of the requests and 30% of the traffic that can be cached. From those results we calculated the lower bound of the cache sizes needed; we found that the lower bound for a cache in the backhaul is 100GB, and 100MB for a cache at the customers' home routers. We then performed a simulation with an LRU cache, in order to validate the results. We found out that a real cache sized according to the ideal values can save around 10% less traffic and requests compared to the ideal cache, while a cache sized 10 times bigger achieves the theoretical performance.

Finally in Chapter 5 we dealt with the problem of scaling up the analysis of the traces using a Hadoop. We compared the performance and the limitations of a classical

C implementation with a Hadoop implementation of the analytics, both using the PIG query language and using directly MapReduce. We found out that the most scalable solution for our use case is using MapReduce directly.

6.2 Conclusions

The main result of this thesis is that caching in the access and backhaul network yields clear advantages to ISPs, customers and content providers, and it is feasible with current technology.

The traffic reduction in the access network translates directly into a reduction of outgoing traffic, and therefore it represents a concrete and obvious saving for the operators. But the most important saving is in the backhaul link from the access network to the core. Those links are usually the bottleneck in both fixed and mobile networks, and very often it's not technically or economically feasible to upgrade them once they saturate. A traffic reduction on the backhaul links would therefore free up bandwidth that can be then used to provide access to more customers, thus benefiting both the additional customers that will have access, and the providers that will see additional revenue with the same infrastructure. The customers will also have a gain in speed due to the fact that the requests will be serviced closer to them. The content providers will see their content delivered faster and to an increased amount of customers, and at the same time less traffic will actually hit their servers, thus increasing the performance of the requests that couldn't be cached.

Another very important and striking conclusion that we can draw is that caching in the access network is already feasible with current technology. 100 GB of fast DRAM are relatively cheap and can fit in any small rack-mount server, and 100 to 1000 MB of RAM can be easily built into a home router. Some home routers actually already come with that much ram, so in that case it would be just a matter of modifying the firmware so as to allow caching.

6.2.1 Transparent Web caching and encryption

As mentioned in Sec.5.1.2, the usage of encryption in the Internet is growing steadily and is also endorsed by recent IETF IAB recommendations. HTTP 2.0 draft [76] currently under discussion in the HTTPbis IETF working group specifies encryption by default by using TLS 1.2 and following versions [77]. It is out of scope of this thesis to fully elucidate the trade-offs of using encryption in today's Internet, for which we refer the reader to [78]. Yet, some architectural considerations are worth sharing concerning the technical challenges that an increasingly encrypted Internet will bring, and that CCN can gracefully solve.

TLS provides communications security for client/server applications and encrypts everything included in the TCP byte stream. The encryption service provided by TLS is not compatible with proxy or transparent caching, which is however a very important service successfully deployed to reduce bandwidth consumption in many network locations. Caching non encrypted Web traffic in proxies or transparent appliances is today implemented and optimised by using HTTP almost as a transport layer. An HTTP datagram has also been proposed in [1] to effectively implement almost all the functionalities CCN provides, with the exception of data encryption and security in general. It is clear that encrypted Web traffic, using TLS, can be cached only in the end points, i.e. the client Web browser, or application, and in content provider appliances. Of course this latter end point can be distributed in a CDN which manages the encryption on behalf of the content provider. Therefore caching encrypted Web traffic cannot be implemented as a transparent network primitive because it would always require the sender to delegate encryption to a third-party, e.g. a CDN.

6.3 Perspectives

6.3.1 Encryption and TLS/SSL

The significant potential gains we have measured do not apply to encrypted web applications (15% of the traffic in the first dataset, 30% in the second, and steadily increasing in time) which obviously cannot be transparently cached. As previously stated, encrypted Web traffic can be cached only at the end points, unless encryption is delegated to third-parties like CDN operators. A minimum level of cooperation is required to guarantee inter-networking of communications primitives which are based on delegation. Datagram based packet-switched networks make use of delegation for data forwarding and routing – but other services like name resolution, as provided by the DNS, do require delegation as well. We believe that in-network data caching is an additional transparent network primitive that cannot be implemented without guaranteeing the required level of security and interoperability that TLS provides.

The clear solution is to use ICN, and in particular the CCN implementation, as the best fit technology to address all the drawbacks of currently available workarounds and providing caching as a transparent network primitive.

6.3.2 Mobile traffic

Another interesting development would be to measure the cacheability of mobile traffic, to assess the feasibility and performance of caches placed on the base stations. Due to its nature, mobile traffic behaves very differently from non-mobile one; in particular the amount of interrupted or incomplete requests is expected to be significantly higher. In order to perform such analysis, the capture software would probably need to be extended, and surely the current hardware setup wouldn't be adequate. Bureaucratic and logistical problems are also expected, since base stations are more difficult to reach.

6.3.3 Further improving the analysis

There are many more improvements that could be made to the analysis, some also requiring improvements in the capture tool.

Proper state tracking of cookies would improve the accuracy of the analysis. This would require adding some fields in the output of the capture tool to provide the value of the cookies (or a hash thereof), and proper state tracking of cookies will doubtlessly increase the complexity of the analysis.

Another improvement could be to consider the caching directives when performing the LRU simulations. HTTP servers sometimes provide headers in the reply that indicate whether the object should be cached or not. This also requires to add some fields in the output of the capture tool; the amount of additional processing needed would be very low.

We have seen that properly accounting for partial content downloads increases the accuracy of the analysis. Some websites and services, especially video streaming, perform chunking on the application level, embedding the requested object ranges in the URL. Since the base URL is different, the tool considers it as separate objects, whereas they should be considered as partial requests. An application-specific URL analyser would be needed to process the URL and extract the actual ranges; the additional load on the analysis step would be minimal, since range requests are already supported.

Appendices

Appendix A

HACkSAw configuration options

This appendix explains the available configuration options for HACkSAw. The name and possible values of each option is provided, together with a description of the functionality it affects. Unspecified values are assigned a default value, also provided here. Parameters specified on the commandline take precedence over the content of the configuration file. All numbers are to be specified as integer numbers.

Name	**Type**	**Default**	**Description**
`connection_timeout`	*minutes*	15	general timeout for closing inactive connections (0 = never close)
`ip.connection_timeout`	*minutes*	*connection_timeout*	timeout for IP connections
`tcp.connection_timeout`	*minutes*	*ip.connection_timeout*	timeout for TCP connections
`udp.connection_timeout`	*minutes*	*ip.connection_timeout*	timeout for UDP connections (streams)
`anon_ips`	*boolean*	yes	anonymise (hash) IP addresses and MAC addresses
`anon_ips.salt`	*integer*	0	salt to use when hashing to anonymise, 0 means random
`tcp.hashtable`	*integer*	1048576	size in elements of the TCP flow hashtable
`udp.hashtable`	*integer*	1048576	size in elements of the UDP flow hashtable

`tcp.enabled`	*boolean*	yes	enable processing of TCP
`udp.enabled`	*boolean*	yes	enable processing of UDP
`http.etags.as_id`	*boolean*	yes	append the value of ETAGs (when available) when calculating the object IDs
`http.hash_urls`	*boolean*	yes	hash HTTP URLs (**things will break if disabled**)
`http.use_dns_info`	*boolean*	false	Use and correlate information gathered by the DNS dissector in the HTTP log
`hacksaw.log`	*path*	/var/log/hacksaw.log	log file, with status and debug information
`hacksaw.daemon`	*boolean*	no	start HACkSAw as a daemon
`hacksaw.autorestart`	*boolean*	no	autorestart the daemon if it crashes
`hacksaw.configfile`	*path*	/etc/hacksaw.conf	configuration file to use
`nthreads`	*integer*	1	number of concurrent processing threads
`input`	*string*	eth0	input file name, interface, or device node
`input.type`	*string*	pcap	input type. e.g. `pcap`, `pcap_file`, `dag`
`output.path`	*path*	.	path or base directory for the output
`output.type`	*string*	file	output type. e.g. `file`, `file.byend`
`output.slotsize`	*seconds*	1 day	slot size of the log files
`input.dag.mem`	*MiB*	512	total system RAM to allocate to the DAG card
`input.dag.path`	*path*	/usr/local/bin/	path where to find DAG binaries
`input.dag.streams`	*string*	hat	how to distribute packets in streams. (`hat`,`ports`)

`input.dag.stream`N`.ports`	*string*	list of interfaces to put in the Nth stream, without separators, e.g. `01`

Appendix B

HACkSAw output files

This appendix describes the contents of the output files generated by HACkSAw. A short description is provided to explain the type of connection logged in each file, and then the fields are presented in the same order as they appear in the file, indicating the type of the value and the meaning.

Many fields have similar formats, so here is a description of the value formats used throughout all the files.

Number An integer number, in base 10.

Hex An integer number, in base 16.

Timestamp Timestamp in UNIX time (number of seconds since Jan 1st 1970 UTC), with microsecond accuracy.

Hash A 16-digits hexadecimal number, produced with the FNV1a hashing algorithm.

IP An IP address, in dotted-decimal format.

MAC A MAC address, in the traditional hex format with colons.

Port A decimal number between 1 and 65535 representing a port number.

Conn-ID A string in the format `Number:Number`, representing a unique connection ID. The first number is the ID of the thread, while the second is the unique connection ID inside that thread. In case a line is logged in the output file with no associated connection, the second number will be 0.

Bitfield[n] A string of length n, each character indicating a binary value. The meaning of the characters is explained case by case.

String A string, can not contain spaces, usually shorter than 30 characters.

Text A string, can contain spaces and can be very long.

An asterisk ($*$) next to the type indicates that the field can be empty, in which case it will contain a dash (`-`) instead of the value.

B.1 log_tcp_complete

This file logs all observed TCP connections, including incomplete ones. Since no text fields are present, the fields are delimited by spaces.

Name	Type	Description
Conn-ID	*Conn-ID*	TCP Connection ID, used to correlate this connection with other L7 requests logged in other files
Protocol	*String*	L7 protocol detected, `?` if unknown
Client IP	*IP*	IP address of the source of the connection
Client Port	*Port*	Port number of the source of the connection
Server IP	*IP*	IP address of the destination of the connection
Server Port	*Port*	Port number of the destination of the connection
Client Packets	*Number*	Number of packets sent by the client
Server Packets	*Number*	Number of packets sent by the server
Client Bytes	*Number*	Number of bytes sent by the client
Server Bytes	*Number*	Number of bytes sent by the server
Start time	*Timestamp*	Timestamp of the first packet of the connection
Handshake time	*Timestamp*	Timestamp of the SYN/ACK packet, when the TCP three-way handshake is over
End time	*Timestamp*	Timestamp of the last packet of the connection

B.2 log_http_complete

This file logs all completed or interrupted HTTP connections observed. Each HTTP transaction (request/reply pair) is logged in a separate line. A Connection ID is provided to correlate requests belonging to the same TCP connection, and to correlate with the data in the `log_tcp_complete`. Since some of the fields can contain spaces, the fields of this file are separated by tabs.

Name	Type	Description
Client MAC	*MAC*	MAC address of the source of the connection

Dest MAC	*MAC*	MAC address of the destination of the connection, or of the first hop towards the destination, if the destination is not link-local
Client IP	*IP*	IP address of the source of the connection
Client Port	*Port*	Port number of the source of the connection
Server IP	*IP*	IP address of the destination of the connection
Server Port	*Port*	Port number of the destination of the connection
Conn-ID	*Conn-ID*	TCP Connection ID, used to correlate the requests from this file with the ones in `log_tcp_complete`
Version	*String*	HTTP version string (e.g. `HTTP/1.1`)
Method	*String*	HTTP method (e.g. `GET`)
Request time	*Timestamp*	Timestamp of the first packet of the HTTP request
Request headers	*Number*	Size of the request headers, in bytes
Object	*Hash*	Hash of the concatenation of object URL and, if available, ETAG
ETAG	*String**	Indicated the presence of the ETAG header
Cookies	*Bitfield[2]*	`c/-` client side cookies present/not present `s/-` server side cookies present/not present
Reply code	*Number*	HTTP response code (e.g. `404`)
Reply headers	*Number*	Size of the reply headers, in bytes
Content-Length	*Number**	Object size as advertised by the `Content-Length` header
Actual length	*Number*	Object size as observed
Range	*Text**	Raw value of the `Range` header, in case of range requests
Reply headers time	*Timestamp*	Timestamp of the first packet of the HTTP reply
Reply content time	*Timestamp*	Timestamp of the first packet of the HTTP reply containing the requested object. Will be 0 if no object is returned.
Reply end time	*Timestamp*	Timestamp of the last packet of the HTTP reply containing the requested object. Will be 0 if no object is returned.
Transfer-Encoding	*String**	Content of the `Transfer-Encoding` header

Hostname	*String*	Content of the `Host` header, or the IP address of the server if no `Host` header is present
Content-Type	*String**	MIME type of the returned object, if any, as indicated by the `Content-Type` header
Client headers	*Hex,String*	Hexadecimal bitfield. Each bit represents the presence of a specific frequently used header in the client request. Headers not present in the list are concatenated to the value, separated by commas.
Server headers	*Hex,String*	Hexadecimal bitfield. Each bit represents the presence of a specific frequently used header in the server reply. Headers not present in the list are concatenated to the value, separated by commas.
DNS request	*Timestamp*	Timestamp of the DNS request used to resolve the domain for this connection. Only present if DNS logging is active, DNS result sharing is active, and DNS request were performed within a reasonable (configurable) timeframe. It will be 0 otherwise.
DNS reply	*Timestamp*	Timestamp of the DNS reply used to resolve the domain for this connection. Only present if DNS logging is active, DNS result sharing is active, and DNS request were performed within a reasonable (configurable) timeframe. It will be 0 otherwise.
User-Agent	*Text*	Content of the `User-Agent` header. Can be very long.

B.3 log_udp_complete

This file logs all observed UDP sessions, including incomplete ones. Since no text fields are present, the fields are delimited by spaces.

Name	**Type**	**Description**

Conn-ID	*Conn-ID*	UDP Connection ID, used to correlate this connection with other L7 requests logged in other files
Protocol	*String*	L7 protocol detected, `?` if unknown
Client IP	*IP*	IP address of the source of the connection
Client Port	*Port*	Port number of the source of the connection
Server IP	*IP*	IP address of the destination of the connection
Server Port	*Port*	Port number of the destination of the connection
Client Packets	*Number*	Number of packets sent by the client
Server Packets	*Number*	Number of packets sent by the server
Client Bytes	*Number*	Number of bytes sent by the client
Server Bytes	*Number*	Number of bytes sent by the server
Start time	*Timestamp*	Timestamp of the first packet of the connection
End time	*Timestamp*	Timestamp of the last packet of the connection

B.4 log_dns_complete

This file logs all observed DNS requests, including incomplete ones. Since no text fields are present, the fields are delimited by spaces.

Name	**Type**	**Description**
Thread-ID	*Number*	Thread ID, used for debugging purposes
Client IP	*IP*	IP address of the source of the connection
Client Port	*Port*	Port number of the source of the connection
Server IP	*IP*	IP address of the destination of the connection
Server Port	*Port*	Port number of the destination of the connection
Transaction ID	*Hex*	Transaction ID as per DNS protocol
Request time	*Timestamp**	Timestamp of the DNS request
Opcode	*Number*	DNS request opcode
Request Flags	*Bitfield[7]*	`R/Q` packet is a response/query
		`A/-` server is authoritative/or not
		`T/-` is truncated/or not
		`r/-` recursion is wanted/or not
		`R/-` recursion is available/or not
		`a/-` reply is authenticated/or not
		`u/-` non authoritative replies accepted/or not

Request Questions	*Number*	Number of question records in the question section of the request
Request Answers	*Number*	Number of answer records in the answer section of the request
Request Authorities	*Number*	Number of name servers records in the authorities section of the request
Request Additionals	*Number*	Number of additional records in the additional section of the request
Request Class	*String*	DNS class of the request. Usually INET.
Request Types	*String*	Types of record requested. It is a comma separated list of all the record types present in the question records
Request IPv4	*String**	List of domains whose IPv4 address is requested (record type A)
Request IPv6	*String**	List of domains whose IPv6 address is requested (record type AAAA)
Request MX	*String**	List of domains whose mailserver address is requested (record type MX)
Request NS	*String**	List of domains whose NS addresses are requested (record type NS)
Reply time	*Timestamp**	Timestamp of the DNS reply
Opcode	*Number*	DNS request opcode
Reply Code	*Number*	DNS reply code
Reply Flags	*Bitfield[7]*	Same meaning as the *Request flags* field
Reply Questions	*Number*	Number of question records in the question section of the request
Reply Answers	*Number*	Number of answer records in the answer section of the request
Reply Authorities	*Number*	Number of name servers records in the authorities section of the request
Reply Additionals	*Number*	Number of additional records in the additional section of the request
Reply Class	*String*	DNS class of the request. Usually INET.
Reply Types	*String*	Types of record requested. It is a comma separated list of all the record types present in the question records

Reply IPv4	*String**	List of IPv4 addresses whose resolution was requested
Reply IPv4 TTL	*String**	List of TTL values for the IPv4 addresses in the reply
Reply IPv6	*String**	List of IPv6 addresses whose resolution was requested
Reply MX	*String**	List of mailserver domains for the requested domain
Reply NS	*String**	List of DNS names or addresses for the requested domain

Appendix C

Presentation of the results

The web-based presentation infrastructure, described in this appendix, was developed by Wuyang Li, and not by the author, at the same time this PhD was taking place; it is included here for reference and completeness. The presentation infrastructure is a web-based system that allows authorised people to have fast and easy access to the graphs of the cacheability and traffic reduction statistics calculated using Hadoop.

Workflow

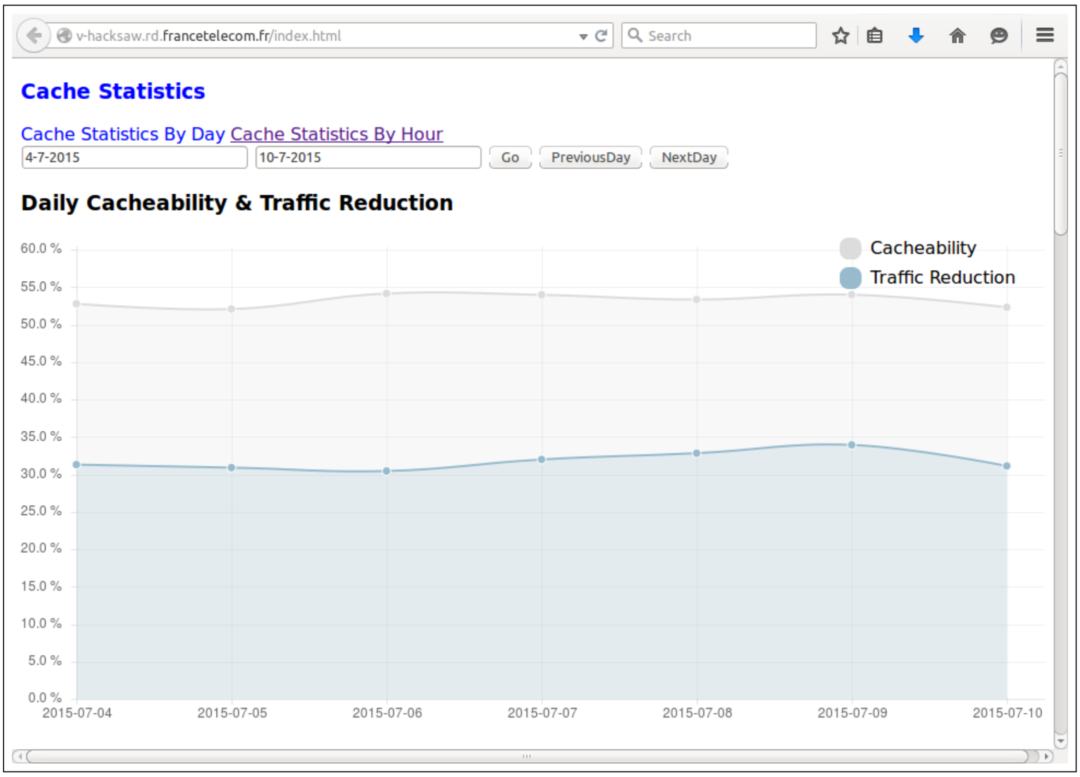
As illustrated in fig.1.2 in Chapter 1, the workflow is rather straightforward: the data is collected by the probe, sent to the Hadoop cluster for analysis, and then sent to the presentation website for display. The website is necessary to have a rapid view on the results of the tool, to check that no data is missing due to network problems on the (slow) control link, and to extract nice graphs to be used in scientific publications. Figure C.1 shows a screenshot of the main page of the website. The controls to select the time granularity and the dates to show are also visible.

The data is inserted directly into a database and the webserver performs some database queries to extract the relevant values to show when interrogated by a web browser.

Technologies

The web server used is Node.js, as it provides superior performance in presence of bursts of requests, such as the ones generated by the frontend. It is also easier to interface client-side javascript libraries with a javascript server.

The database used is MySQL, since only a few simple tables are needed. On the



v-hacksaw.rd.francetelecom.fr/index.html

Cache Statistics

Cache Statistics By Day Cache Statistics By Hour

4-7-2015 10-7-2015

Daily Cacheability & Traffic Reduction

Figure C.1: Screenshot of the first page of the webpage.

other hand, those tables need to be accessed quickly, and at arbitrary positions. The libraries used also expect to find the data in a database.

The webpage itself is written in HTML5, and uses the Chart.js and GoogleChart libraries to display the graph. The interaction between the webpage itself and the backend is performed through AJAX (Asynchronous Javascript And XML), in order to provide a smooth and modern experience to the user.

Frontend and backend

The backend, written in javascript, makes heavy use of asynchronous event-driven callbacks, in order to improve the response times and the overall performance. The MySQL database has a table for each available time granularity (one day and one hour); each row represents all the relevant statistics for that timeslot, indexed by timestamp. The server software itself is composed of 4 different javascript files implementing the main module, the HTTP server, the SQL request handler, and the glue to connect everything together.

The frontend of the system is the webpage, which interacts with the server using asynchronous AJAX requests, with the requested datapoints for each of the statistics shown in the page. Changing the dates causes new AJAX requests to be sent to the server, and the graph is updated smoothly. For convenience, there are buttons to go one day back or forward.

The statistics displayed, in four separate graphs, are:

— Cacheability

— Traffic reduction

— Virtual Cache Size

— Share of HTTP vs HTTPS connections

— Distribution of the content popularity

Cacheability and Traffic reduction are plotted in the same graph (as shown in the screenshot in Fig.C.1), since they use the same scale (percentage), while the other statistics all have distinct graphs.

Bibliography

[1] L. Popa, A. Ghodsi, and I. Stoica, "HTTP as the narrow waist of the future Internet," in *ACM SIGCOMM Workshop on Hot Topics in Networks (HotNets'X)*, 2010.

[2] C. V. N. Index, "Forecast and methodology, 2014-2019 white paper," tech. rep., Technical Report, Cisco, 2015.

[3] `http://blog.netflix.com/2014/04/the-case-against-isp-tolls.html`.

[4] `http://www.jet-stream.com/technology-overview/`.

[5] `http://www.altobridge.com/`.

[6] `https://www.netflix.com/openconnect`.

[7] V. Jacobson, D. Smetters, J. Thornton, and al., "Networking Named Content," in *Proc. of ACM CoNEXT*, 2009.

[8] F. Huici, A. Di Pietro, B. Trammell, J. M. Gomez Hidalgo, D. Martinez Ruiz, and N. d'Heureuse, "Blockmon: A high-performance composable network traffic measurement system," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 4, pp. 79–80, 2012.

[9] `http://hadoop.apache.org/`.

[10] `http://stratosphere.eu/`.

[11] `https://hama.apache.org`.

[12] `http://giraph.apache.org/`.

[13] `http://spark.apache.org/`.

[14] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[15] Y. Lee, W. Kang, and H. Son, “An internet traffic analysis method with mapreduce,” in *Network Operations and Management Symposium Workshops (NOMS Wksps), 2010 IEEE/IFIP*, pp. 357–361, April 2010.

[16] T. Samak, D. Gunter, and V. Hendrix, “Scalable analysis of network measurements with hadoop and pig,” in *Network Operations and Management Symposium (NOMS), 2012 IEEE*, pp. 1254–1259, April 2012.

[17] Y. Lee and Y. Lee, “Toward scalable internet traffic measurement and analysis with hadoop,” *SIGCOMM Comput. Commun. Rev.*, vol. 43, pp. 5–13, Jan 2012.

[18] J. Yang, S. Zhang, X. Zhang, J. Liu, and G. Cheng, “Characterizing smartphone traffic with mapreduce,” in *Wireless Personal Multimedia Communications (WPMC), 2013 16th International Symposium on*, pp. 1–5, June 2013.

[19] R. Fontugne, J. Mazel, and K. Fukuda, “Hashdoop: A mapreduce framework for network anomaly detection,” in *Computer Communications Workshops (INFOCOM WKSHPS), 2014 IEEE Conference on*, pp. 494–499, IEEE, 2014.

[20] K. Singh, S. C. Guntuku, A. Thakur, and C. Hota, “Big data analytics framework for peer-to-peer botnet detection using random forests,” *Information Sciences*, vol. 278, pp. 488–497, 2014.

[21] M. Spina, D. Rossi, M. Sozio, S. Maniu, and B. Cautis, “Snooping wikipedia vandals with mapreduce,” in *IEEE ICC*, (London, UK), Jun 2015. keyword=measurement.

[22] A. Ghodsi, S. Shenker, T. Koponen, A. Singla, B. Raghavan, and J. Wilcox, “Information-centric Networking: Seeing the Forest for the Trees,” in *Proc. of ACM HotNets-X*, 2011.

[23] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, “Less Pain, Most of the Gain: Incrementally Deployable ICN,” in *Proc. of ACM SIGCOMM*, 2013.

[24] G. Carofiglio, M. Gallo, L. Muscariello, M. Papalini, and S. Wang, “Optimal Multipath Congestion Control and Request Forwarding in Information-Centric Networks,” in *Proc. of IEEE ICNP*, 2013.

[25] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, “Scalable mobile backhauling via information-centric networking,” in *Proc. of LANMAN*, 2015.

[26] G. Carofiglio, M. Gallo, and L. Muscariello, “Bandwidth and Storage Sharing Performance in Information Centric Networking,” *Elsevier Computer Networks,*, 2013.

[27] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, “Modeling Data Transfer in Content-Centric Networking,” in *Proc. of ITC23*, 2011.

[28] F. Olmos, B. Kauffmann, A. Simonian, and Y. Carlinet, "Catalog dynamics: Impact of content publishing and perishing on the performance of a lru cache," in *Proc. of ITC26*, 2014.

[29] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: why it matters and how to model it," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 5, pp. 5–12, 2013.

[30] B. Ramanan, L. Drabeck, M. Haner, N. Nithi, T. Klein, and C. Sawkar, "Cacheability analysis of HTTP traffic in an operational LTE network," in *In Proc. of WTS*, 2013.

[31] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park, "Comparison of caching strategies in modern cellular backhaul networks," in *Proc. of ACM MobiSys*, 2013.

[32] G. Carofiglio, M. Gallo, and L. Muscariello, "Bandwidth and Storage Sharing Performance in Information Centric Networking," in *ACM SIGCOMM, ICN Workshop*, 2011.

[33] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, "Modeling Data Transfer in Content-Centric Networking," in *ITC*, 2011.

[34] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," in *ITC*, 2012.

[35] E. J. Rosensweig, J. Kurose, and D. Towsley, "Approximate Models for General Cache Networks," *IEEE INFOCOM*, 2010.

[36] G. Rossini and D. Rossi, "Evaluating ccn multi-path interest forwarding strategies," *Computer Communications*, vol. 36, no. 7, 2013.

[37] R. Chiocchetti, D. Rossi, G. Rossini, G. Carofiglio, and D. Perino, "Exploit the known or explore the unknown?: hamlet-like doubts in icn," in *ACM SIGCOMM, ICN Workshop*, 2012.

[38] R. Chiocchetti, D. Perino, G. Carofiglio, D. Rossi, and G. Rossini, "INFORM: a dynamic interest forwarding mechanism for information centric networking," in *ACM SIGCOMM, ICN Workshop*, 2013.

[39] C. Yi, A. Afanasyev, I. Moiseenko, L. Wang, B. Zhang, and L. Zhang, "A case for stateful forwarding plane," *Computer Communications*, vol. 36, no. 7, 2013.

[40] D. Rossi and G. Rossini, "On sizing ccn content stores by exploiting topological information," in *IEEE INFOCOM, NOMEN Worshop,*, (Orlando, FL), March 25-30 2012.

[41] I. Psaras, W. K. Chai, and G. Pavlou, “Probabilistic in-network caching for information-centric networks,” in *ACM SIGCOMM, ICN Workshop*, 2012.

[42] K. Cho, M. Lee, K. Park, T. Kwon, Y. Choi, and S. Pack, “WAVE: Popularity-based and collaborative in-network caching for content-oriented networks,” in *IEEE INFOCOM, NOMEN Workshop*, 2012.

[43] W. Chai, D. He, I. Psaras, and G. Pavlou, “Cache less for more in information-centric networks,” in *IFIP Networking*, 2012.

[44] G. Rossini and D. Rossi, “Coupling caching and forwarding: Benefits, analysis, and implementation,” in *1st ACM SIGCOMM Conference on Information-Centric Networking (ICN-2014)*, (Paris, France), pp. 127–136, 2014.

[45] A. Araldo, D. Rossi, and F. Martignon, “Design and evaluation of cost-aware information centric routers,” in *ACM SIGCOMM ICN*, 2014.

[46] M. Zink, K. Suh, Y. Gu, and J. Kurose, “Characteristics of YouTube network traffic at a campus network - Measurements, models, and implications,” *Elsevier Computer Networks*, 2009.

[47] B. Ager, F. Schneider, J. Kim, and A. Feldmann, “Revisiting cacheability in times of user generated content,” in *Proc. of IEEE Global Internet Symposium*, 2010.

[48] F. Schneider, B. Ager, G. Maier, A. Feldmann, and S. Uhlig, “Pitfalls in HTTP Traffic Measurements and Analysis,” in *Proc. of PAM*, 2012.

[49] A. Finamore, M. Mellia, Z. Gilani, K. Papagiannaki, V. Erramilli, and Y. Grunenberger, “Is There a Case for Mobile Phone Content Pre-staging?,” in *Proc. of ACM CoNEXT*, 2013.

[50] `http://www.bro.org`.

[51] `http://tstat.tlc.polito.it`.

[52] H. Yu, D. Zheng, B. Y. Zhao, and W. Zheng, “Understanding user behavior in large-scale video-on-demand systems,” in *Proc. of the ACM SIGOPS/EuroSys*, 2006.

[53] M. S. Allen, B. Y. Zhao, and R. Wolski, “Deploying Video-on-Demand Services on Cable Networks,” in *Proc. of ICDCS*, 2007.

[54] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, “Youtube Traffic Characterization: a View From the Edge,” in *Proc. of the ACM SIGCOMM IMC*, 2007.

[55] M. Cha, H. Kwak, P. Rodriguez, Y.-Y. Ahn, and S. Moon, “Analyzing the Video Popularity Characteristics of Large-Scale User Generated Content Systems,” *IEEE/ACM Transactions on Networking*, 2009.

[56] E. G. Coffman, Jr. and P. J. Denning, *Operating Systems Theory.* Prentice Hall Professional Technical Reference, 1973.

[57] H. Abrahamsson and M. Nordmark, "Program popularity and viewer behaviour in a large tv-on-demand system," in *ACM SIGCOMM IMC*, 2012.

[58] A. Bar, P. Casas, L. Golab, and A. Finamore, "Dbstream: an online aggregation, filtering and processing system for network traffic monitoring," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*, pp. 611–616, IEEE, 2014.

[59] S. Sakr, A. Liu, and A. G. Fayoumi, "The family of mapreduce and large-scale data processing systems," *ACM Comput. Surv.*, vol. 46, pp. 11:1–11:44, Jul 2013.

[60] `https://dato.com/products/create/`.

[61] B. Elser and A. Montresor, "An evaluation study of BigData frameworks for graph processing," in *Proc. of the 2013 IEEE International Conference on Big Data (BigData'13)*, (Santa Clara, CA, USA), pp. 60–67, IEEE, Oct 2013.

[62] `http://valgrind.org`.

[63] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Rfc 2616, hypertext transfer protocol – http/1.1," 1999.

[64] B. Ramanan, L. Drabeck, M. Haner, N. Nithi, T. Klein, and C. Sawkar, "Cacheability analysis of http traffic in an operational lte network," in *Wireless Telecommunications Symposium (WTS), 2013*, pp. 1–8, April 2013.

[65] J.-P. Laulajainen, A. Arvidsson, T. Ojala, J. Seppanen, and M. Du, "Study of youtube demand patterns in mixed public and campus wifi network," in *Wireless Communications and Mobile Computing Conference (IWCMC), 2014 International*, pp. 635–641, Aug 2014.

[66] M. Gallo, B. Kauffmann, L. Muscariello, A. Simonian, and C. Tanguy, "Performance Evaluation of the Random Replacement Policy for Networks of Caches," *Elsevier Performance Evaluation*, 2014.

[67] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *Selected Areas in Communications, IEEE Journal on*, vol. 20, no. 7, pp. 1305–1314, 2002.

[68] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical ttl-based cache networks," *Computer Networks*, vol. 65, pp. 212–231, 2014.

[69] W. Gong, Y. Liu, V. Misra, and D. Towsley, “On the Tails of Web File Size Distributions,” in *Proc. of Allerton Conference on Communication, Control, and Computing*, 2001.

[70] R. B. D’Agostino and M. A. Stephens, eds., *Goodness-of-fit Techniques.* New York, NY, USA: Marcel Dekker, Inc., 1986.

[71] I. B. Aban and M. M. Meerschaert, “Generalized least-squares estimators for the thickness of heavy tails,” *Journal of Statistical Planning and Inference*, vol. 119, pp. 341–352, 2004.

[72] S. I. Resnick, “Heavy tail modeling and teletraffic data,” *The Annals of Statistics*, vol. 25, no. 5, pp. 1805–1849, 1997.

[73] H. Dres, L. De Han, and S. Resnick, “How to make a hill plot,” *The Annals of Statistics*, vol. 28, no. 1, pp. 254–274, 2000.

[74] P. R. Jelenković, “Asymptotic approximation of the move-to-front search cost distribution and least-recently-used caching fault probabilities,” *The Annals of Applied Probability*, vol. 9, no. 2, pp. 430–464, 1999.

[75] G. Rossini, D. Rossi, G. Garetto, and E. Leonardi, “Multi-Terabyte and Multi-Gbps Information Centric Routers,” in *IEEE INFOCOM*, 2014.

[76] M. Belshe, R. Peon, and M. Thomson, “Hypertext transfer protocol version 2,” 2014.

[77] T. Dierks and E. Rescorla, “The Transport Layer Security (TLS) Protocol Version 1.2.” RFC 5246 (Proposed Standard), Aug. 2008. Updated by RFCs 5746, 5878, 6176.

[78] D. Naylor, A. Finamore, I. Leontiadis, Y. Grunenberger, M. Mellia, M. Munafò, K. Papagiannaki, and P. Steenkiste, “The cost of the ”s” in https,” in *Proc. of ACM CoNEXT*, 2014.

List of Publications and Submissions

[79] C. Imbrenda, L. Muscariello, and D. Rossi, “Analyzing cacheable traffic in isp access networks for micro cdn applications via content-centric networking,” in *Proceedings of the 1st International Conference on Information-centric Networking*, ICN ’14, (New York, NY, USA), pp. 57–66, ACM, 2014.

[80] C. Imbrenda, L. Muscariello, and D. Rossi, “Analyzing cacheability in the access network with hacksaw,” in *Proceedings of the 1st International Conference on Information-centric Networking*, ICN ’14, (New York, NY, USA), pp. 201–202, ACM, 2014.

[81] C. Imbrenda, W. Li, and L. Muscariello, “Analyzing cacheable traffic for ftth users using hadoop,” in *Proceedings of the 2nd International Conference on Information-Centric Networking*, ICN ’15, (New York, NY, USA), pp. 191–192, ACM, 2015.

[82] C. Imbrenda, L. Muscariello, and D. Rossi, “Revisiting traffic cacheability in the age of information centric networks.” Submitted to IEEE/ACM Transactions on Networking (2nd phase), 2015.

[83] C. Imbrenda, W. Li, L. Muscariello, and D. Rossi, “Scaling up the analysis of traffic cacheability using map-reduce.” Submitted to Elsevier Computer Networks Journal, 2015.