Networking 2013 1569701111

CobWeb: In-Network Cobbling of Web Traffic

†Hitesh Khandelwal, ‡Fang Hao, ‡Sarit Mukherjee, †Ramana Rao Kompella, ‡T.V. Lakshman
†Purdue University, ‡Bell Labs Alcatel-Lucent

***Abstract*—Existing per-flow measurement solutions provide coarse-grained information for ISPs to manage their networks, that are insufficient for many new emerging applications such as reverse billing, where charges associated with accessing a web service are billed back to the web service provider rather than the users. For such applications, it is important to construct the entire *session tree* that represents the transitive closure of all traffic downloaded as a result of a user accessing a given web service. Automated construction of the session tree based on network traffic observation is challenging mainly due to the complex composition of modern web services. In this paper, we present CobWeb, a system that performs automated in-network cobbling and monitoring of web services traffic. Using extensive evaluation using over 700GB of traces, we show CobWeb can achieve good accuracy with low ($< 5\%$) false positive and negative rates.**

I. Introduction

Today, per-flow traffic measurement and application identification are extensively used by ISPs to gain insight into their network operations. These are often done using several specialized monitoring boxes currently in the market [2], [1], that provide extensive per-session, per-application or per-flow usage and performance reports. These boxes also identify traffic to various popular web-sites by time-of-day, region, etc. However, these solutions provide measurements at a very coarse-granularity (*e.g.*, IP address, protocol signature), which is not sufficient for many emerging network measurement tasks such as the *reverse billing* application.

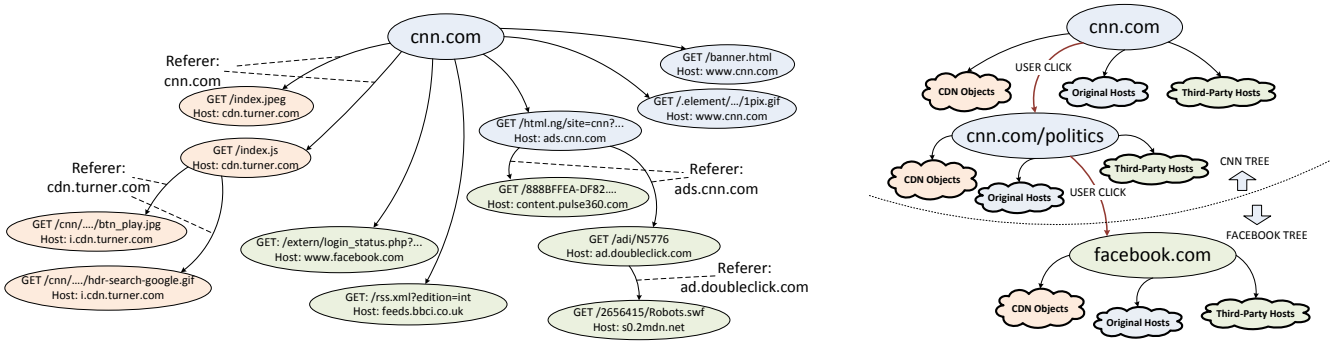
In reverse billing, any charges associated with accessing a web service are billed back to the web service provider rather than to users. It is motivated by the growing shift from flat-rate to tiered-pricing in wireless networks, and in some countries, wired networks as well. Examples include AT&T which permits 250MB data for about $15 and 2GB for $25 per month in the US. Thus, service providers may want to attract customers by providing 'toll-free' access to their services. Another alternate could be a subsidized service model where the ISP may provide access to web-services such as ESPN or CNN for some nominal fee per month.

For such applications, it is important to precisely associate all the traffic that is generated upon each access to a given web service. This entails identifying all the traffic that is due to downloads from the original host for the web service, its content delivery network (CDN), and downloads of embedded objects from third-party services (*e.g.*, advertisements). Unlike per-flow traffic measurement, for this more general problem, it is necessary to construct a *session tree* that represents the transitive closure of all web service accesses that happen as a consequence of accessing a given root web service. Note that the session tree may be quite dynamic and the internal nodes can change across users as well as across time. While we motivated this measurement problem in the context of reverse billing, we believe there are several other contexts where such a capability is useful. To our knowledge, there is no effective solution to this general measurement problem yet, and we are among the first to consider this problem.

Constructing the session tree in the network is challenging for many reasons. Widely used web services are a complex mashup of content from several supporting services including CDNs, third-party advertisement platforms (*e.g.*, ads.doubleclick.com), and other third-party services (*e.g.*, CNN web services using Facebook for friend recommendations). This makes the structure of the session tree difficult to infer. Also, with the inherent flexibility in designing web services, the session tree is rarely static. Moreover, web services are largely personalized—the content served varies with the user even for the same URI. Thus, one cannot use a unique set of URIs to identify a service. Yet another issue is that potentially many different web services are usually co-hosted within a data center (*e.g.*, Akamai) making IP addresses ineffecitve.

In this paper, we describe a system, CobWeb, that automatically performs in-network cobbling of different web services—we use the term "cobbling" for the identification and measurement of all traffic associated with a given web service. We assume that CobWeb has access to both upstream and downstream traffic flows, as it is envisioned to be deployed at the network edge with port mirroring used for access to traffic flows. CobWeb works in two stages. It identifies any supporting CDN used by a web service and then, it identifies all the embedded objects downloaded for that web service. The total data usage consists of all the traffic that is associated with access to this service—be it from the original access, from the CDNs, and from all the related chain of accesses that are triggered by the embedded links. Designing a system that tracks all the traffic belonging to a web service with 100% accuracy is a big challenge given the lack of any uniform methodology or standards in the composition of web services; the mechanisms that we use therefore rely on heuristics that reflect current practices in web service composition.

We evaluate our system based on two web traffic traces in total amount of 739 GB collected from a large university campus network, along with traces that are generated in lab controlled environment for emulating user browsing behaviors. We take 70 web sites from the Alexa top 100 US sites, along with the top 26 popular web sites for users in a large university campus network, as the target web services. Our results show that the system can achieve an average false positive rate of



(a) Constituents of a CNN session

(b) Defining CNN web session

Fig. 1. Subfigure (a) shows a subset of URLs downloaded when a user downloads the base `cnn.com` page. Subfigure (b) shows that clicks to third party websites are not part of the CNN session tree.

3.5% and false negative rate of 4.8% across these 96 web services.

II. Problem Statement

In this section, we start with some preliminaries about web services. We then clearly define our main objective in this paper, argue why the problem is hard, and show that simple solutions do not work well.

Web Service Preliminaries A web browser interacts with a web server by sending HTTP requests to the server, and then receiving response messages back. The page returned in the response to a request to say cnn.com web service may contain many links to other embedded objects as shown in Figure 1(a). For accessing the CNN page, more than 150 additional requests are sent to 24 other hosts to acquire all the additional content. Of course, such numbers may change due to dynamic nature of the content. The hosts that provide content to a web page can be classified into three broad categories: *original hosts*, *CDN hosts* and *third party hosts*. Original hosts are those that belong to the same root domain of the web service, *e.g.*, cnn.com and money.cnn.com. CDN hosts are the servers that are part of the CDNs associated with that main domain (*e.g.*, cdn.turner.com is the main CDN associated with CNN). Third-party hosts provide content such as advertisements, statistics collection, social networking, and so on (*e.g.*, feeds.bbci.co.uk and www.facebook.com for CNN).

Objective Our goal is to 'cobble' an entire *session tree* corresponding to a user accessing a given web service in the network at an ISP edge router, where both directions of user traffic are visible. We define a web session more precisely as follows: (1) All the content downloaded from the original hosts (*e.g.*, *.cnn.com) responsible for the web service. (2) All the content downloaded from CDN servers (*e.g.*, *.cdn. turner.com) the web service uses as part of the session. (3) All the embedded objects automatically downloaded for the web service from any servers, *e.g.*, original hosts, CDNs, or third-party hosts.

Figure 1(b) shows an example navigation of the CNN web page. The root of the session tree starts at cnn.com that involves the browser automatically fetching embedded content from the original hosts, CDN hosts or third party hosts. The user clicks on cnn.com/politics leads to another series of sessions to various hosts and this is again considered part of the session tree since the click leads to a CNN webpage. When a user clicks on a link to a third party website such as facebook.com, we consider it to be outside of the CNN session tree (as shown in the figure).

Though not shown in the example, any clicks to URLs which involve the CDN hosts or original hosts are considered as part of the session tree. For example, if the user clicks on URL cdn.turner.com/xyz.html (a contrived example), it would be considered a part of the session tree. While this example started with cnn.com, a user may directly enter the URL cnn.com/politics into the browser and make that URL the root. However, we do not consider session trees starting directly from a CDN URL as part of the web service since CDNs are known to be shared across different web services. For example, cdn.turner.com may host content from tbs.com.

Why is the problem hard ? The *key difficulty* comes from the fact that routers only observe a stream of HTTP requests but there is no obvious handle one can use to easily isolate the requests that belong to a given session. One simple solution is to, for every web service of interest, simply keep the domain names or IP addresses that it needs to match. Anecdotal evidence suggests that ISPs may be doing exactly this, due to the lack of a more compelling alternative. While such an approach may have worked 10-15 years ago when web services were very simple, *e.g.*, a few servers would serve static HTML content, unfortunately this approach will not work well today due to their more complex composition. For instance, many web services involve fetching objects from common domains. For example, the main cnn.com and nytimes.com web pages include an embedded request to facebook.com. Thus, the request to Facebook needs to be classified as part of CNN or NYTimes or even just Facebook depending on the overall context, making it difficult to come up with a blanket rule for all websites. IP addresses unfortunately are not helpful either since the same web servers (*e.g.*, Akamai) may be hosting content from different providers.

In certain settings, if we have cooperation from a given web service provider, the problem may become easier. We however cannot assume that such cooperation is the norm always.

Indeed, for the reverse billing application, the relationship may only be between the customer and the ISP, with no cooperation from website content provider. (Thus, for generality, in this paper, we assume no such cooperation.) We cannot also assume any client cooperation since it is too intrusive an approach to run a special agent on the client side machine.

III. CobWeb Design

In this section, we describe the design of CobWeb, a system for in-network cobbling of web service traffic. We first present an overview of our approach, and then describe the heuristics that form the basis for CobWeb. We assume that both directions of the traffic can be observed by CobWeb.

A. Overview

In our approach, we mainly leverage a key field within the HTTP headers, namely the 'Referer' field, which most browsers today set. This field mainly indicates the (previous) page that referred to the current page. For example, when a user clicks on www.cnn.com/politics URL on the www.cnn.com/US web page, the corresponding GET request will contain www.cnn.com/US as the Referer. Note that the Referer field contains both a referrer host (e.g., cnn.com) and referrer URI (e.g., /US) should it be present. We leverage the Referer field to keep track of the navigation chains to identify the roots of the session trees.

While using the Referer field, one can form an association between two different webpages A and B if A led to B, it is not always easy to establish whether B was a result of the user clicking on A, or an automated download. The reason why this is important is that automated downloads need to be counted as part of the actual web service, even if it is to third party domains, *i.e.*, non-origin domains. On the other hand, user clicks to third party domains, that start a new session tree (as we discussed in Figure 1(b)). However, sometimes a user can click on an associated CDN domains (*e.g.*, turner.com for CNN), which must be considered as part of the session tree. Thus, to make this differentiation, it is important to first establish the set of CDNs for a given domain, after which we need methods to differentiate between the embedded downloads and clicks to third party sites. Note however, all accesses to the CDN cannot be considered as part of that particular web service, since the same CDN may host multiple web services.

Our overall approach therefore consists of the two basic steps: *CDN detection* and *Embedded object detection*. The CDN detection step is an offline process that involves identifying the CDN (or supporting) domains that play an important supporting role for delivering a given web service. We expect that for web services of interest, we separately track their associated CDNs (which rarely change) and incorporate them into the cobbling process. For embedded object detection, which is an online process, the goal is to identify the set of embedded objects fetched as part of a given web page as opposed to those that are retrieved due to user clicks. The key metric used here to distinguish between the two types of retrievals is that embedded object retrieval has much less 'think time' than user-click based retrieval since the embedded objects are automatically fetched by the web browser. We also use the fact that some embedded objects have standard file-types such as javascript (extension .js, .json) that are not usually associated with objects retrieved by user clicks.

Given the importance of the Referer field in our approach, one could argue that it is easy to disable the Referer field since many modern browsers provide the appropriate settings anyway. In most modern browsers, however, the Referer field is by default turned on; very few people even bother to turn it off (or are even savvy enough to turn it off). If indeed, the Referer field were turned off completely by a majority of the users, the whole multi-billion dollar Internet advertisement industry would crumble, since they heavily rely on the Referer field for tracking the source of their clicks. Thus, companies such as Google/Microsoft have incentive to keep the Referer field on in Chrome/IE browsers to support their online advertising businesses. We discuss our heuristics next.

B. CDN Detection

The goal of CDN detection is to identify supporting CDNs (if any) for a web service. In the example of Figure 1(a), the focus for this would be the left portion of the tree, i.e., requests with host *.cdn.turner.com that belong to the CDN for CNN. One method for CDN detection is to monitor web request traffic at a network edge router (*e.g.*, campus gateway or ISP border router), and use it to identify the domain that has delivered the most amount of traffic to clients when the pages are downloaded.

Implementing this idea is not straightforward since we still need to identify, from the monitored traffic, the total traffic to a web site (which is the cobbling problem that we started with). We first focus on the portion of the traffic that can be clearly identified—the HTTP GET/POST requests with referer URL belonging to the root domain. For example, to detect CDN for cnn.com, we first look at all requests where the request has its referer host as cnn.com or sub-domain of cnn.com such as money.cnn.com. This is the traffic for downloading the embedded pages of the root domain (*e.g.*, embedded content for main cnn.com web page) and the traffic for accessing pages when user navigates away from the page (*e.g.*, user clicks on nytimes.com on cnn.com page). Note that in the second case, only the first GET or POST request has referer as cnn.com. The rest of the requests have nytimes.com or subsequent pages as referer. As long as the traffic for accessing the external web sites, in the second case, does not exceed the traffic for accessing the CDN for cnn.com, it will not affect the result of CDN detection.

Another issue is that web services may use multiple CDNs. For example, cnn.com uses both Level 3 and Akamai CDNs. Fortunately, in many cases, we find that the CDN host contained in HTTP requests is an alias of the canonical name (CNAME) of the actual server. For example, cnn.com has i,z.cdn.turner.com for different types of content. i.cdn.turner.com is an alias for CNAME cdn.cnn.com.c.footprint.net and is owned by Level 3, z.cdn.turner.com is an alias for CNAME z.cdn.turner.

com.edgesuite.net and is owned by Akamai. Use of this kind of aliasing allows web pages to be not tied to any particular CDN provider. Given the structure of the CDN aliases, we can detect the "CDN domain" (*e.g.*, cdn.turner.com) instead of specific CDN host (*e.g.*, i.cdn. turner.com). Our algorithm looks at different levels of the host domain, and tries to aggregate them. For example, level-1 (top-level) domain of i.cdn.turner.com is com, level-2 domain of that is turner.com.

CDN Detection Algorithm We start with traffic monitored at the network edge router. The algorithm starts by looking for all the HTTP GET requests that contain the main host domain as the referer (*e.g.*, cnn.com or ads.cnn.com or money.cnn.com for CNN). Let this total number be n. We also compute the break down of these requests individually to each and every host h (denoted by n_h). For example, if x and y GET requests with referer as cnn.com were made to disqus.com and cdn.turner.com, respectively, we denote n_{disqus}= x and $n_{cdn.turner}$=y. Note that all counts are in number of bytes. Out of all hosts h, we pick the host with maximum portion of traffic (n_h/n) as the top host.

Suppose we identify i.cdn.turner.com as the top host in this step, we then try the next aggregated level of the top host domain: cdn.turner.com by repeating the counting procedure for level-3 domains. We can continue this procedure for further aggregated levels to get the top domain at each level. Starting from the lowest level (most aggregated) $l = 3$, we select the top level-3 domain as the CDN domain if the difference between the proportion of traffic for the top level-3 domain and the top level-4 domain is above a pre-set threshold t ($r_3 - r_4 > t$). In our system we choose t to be 5%. Otherwise we do not use the level-3 domain, and check the next level $l = 4$. We select level-4 if $r_4 - r_5 > t$. We continue this process until either we find a level l such that $r_l - r_{l+1} > t$ or l is the full host domain. We then select top level-l domain as the CDN domain.

Special cases For less popular web sites that do not use any CDNs, the main domain usually covers majority of the traffic anyway. However, we need to handle a web site that heavily uses services from some third party sites. For example, we found that reddit.com is a popular site (in the Alexa top 50 sites in the US) that relies on imgur.com for hosting images, although imgur.com is neither a CDN nor the main supporting domain owned by reddit.com. Given that our algorithm selects the most heavily referred site as the CDN, it will end up picking imgur.com as the CDN for reddit.com, which is not true even though its presence may be vital to the particular web service. The reason we need to differentiate imgur.com from CDN sites is that the clicks beyond the embedded requests at imgur.com should not be part of reddit.com service, unlike for CDN sites. In that sense, imgur.com should be considered similar to a third-party host, where embedded object accesses from the origin web pages are considered part of the web service in question while user clicks are not.

One other issue is that multiple web sites may claim the same CDN as its own CDN. For example, both cnn.com and adultswim.com use cdn.turner.com as the CDN. This is acceptable for our method, since we can trace back to the origin domain through the chain of referer fields and separate out the requests originating at different origin domains.

C. Embedded Object Detection

Besides the traffic from the origin domain and CDN (or main supporting domain), there is also traffic for downloading embedded content from third party web sites. This includes objects such as www.facebook.com/extern/login... and ad.doubleclick.com/adi/... in the example shown in Figure 1(a). In this section, we investigate two methods for detecting requests for fetching such objects—one based on the file-type extensions and the other based on timing.

Classification based on file-types: Our first observation is that certain file-types are almost always embedded. This includes css, js, swf, ico, json, and xml. Download of such files is always triggered automatically by the download of another (embedding) web page since such files are not useful on their own. Our preliminary inspection of the trace collected at a large university gateway shows that 15% in terms of requests and 11.3% in terms of bytes of all the HTTP traffic are for such files. In terms of percentage, they cover a smaller portion of traffic than the CDN, but still significant in terms of volume especially for services that use these embedded objects more frequently. In addition, the almost certainly embedded nature of these objects makes it an accurate classification rule, and is also easy to implement. It also helps cross-checking with other heuristics as we discuss next.

Classification based on timing: Our second observation is that the embedded objects are downloaded shortly after its referer page (called the base page). The time interval between the two downloads should typically be shorter than the time it takes for a user to browse a given web page and then click on a URL in the page to navigate to a different page. For convenience, we define the time interval between downloading the base page and the embedded links as *think time*. More precisely, we define think time $T_{think} = T_G - T_R$, where T_G is the time at which the GET/POST request for the embedded page has been sent, and T_R is the time instant at which the last response packet of the corresponding referer page arrived. For example, the think time for URL cdn.turner.com/index.jpeg in Figure 1(a) is the interval between the time when the last response packet for its referer cnn.com is received and when the GET request for cdn.turner.com/index.jpeg is sent. Note that it is possible to have $T_{think} < 0$ since browsers can start downloading the embedded URL even before it finishes downloading the entire base page. Intuitively, think time is the time it takes for the browser to process the web page, extract any embedded URLs, and then send subsequent requests to download these embedded objects.

Naive timing heuristic: We may use the following naive heuristic: A session is classified as embedded URL download if $T_{think} < T_{thresh}$, where T_{thresh} is a timing threshold, *e.g.*, 1 second. Unlike the file-type heuristic, the timing heuristic can generate both false negatives (missing requests part of the target

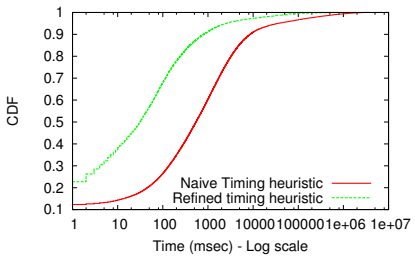


Fig. 2. Think time distribution for embedded object downloads with both naive and refined heuristics.

web service) and false positives (including requests after the user navigates to third party web pages) depending on the value of T_{thresh}. To understand how practical the timing heuristic is, we take advantage of the file-type heuristic described in previous section.

Results from a real packet trace. We calculate the think time for all embedded file downloads based on the file-type heuristic in a full HTTP trace we collected at the large university gateway. Figure 2 shows the think time distribution. We observe that think time varies across a wide range. Although about 60% of think time falls below 1 second, 10% of think time is above 10 seconds. If we naively set the T_{thresh} to 10s, we will be able to capture almost 90% of all the embedded objects. But, this has the negative effect of increasing the false positives, since 10 seconds is sufficient time for a user to click on a third party link, which will then be classified as an embedded object. Similarly, keeping threshold too low will cause missing many embedded objects. Finding a fixed threshold that will work for a large number of web-services is not easy.

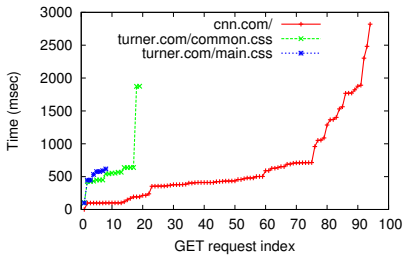


Fig. 3. Think time for multiple embedded objects with the same referer

Refined timing heuristic: To better understand why the browser think times are sometimes exceedingly long, we inspect the time sequence of web page downloads more carefully. Figure 3 shows the timing for downloading multiple embedded URLs following the referer URL cnn.com/XXX. The X-axis shows the index of the URLs sorted according to the time when the GET/POST request is sent. We observe that the think time increases almost linearly though the requests towards the end are spaced farther apart than at the beginning. The increased spacing is because third-party embedded objects or advertisements are among the last to be requested and they take more time to load as well. In addition, Browser limits on parallelism causes sequential access of embedded URLs leading to this phenomenon.

Examination of the figure further reveals that the time offset when a GET request for an embedded object is made, relative to the request for the base page, is proportional to the number of GET requests for embedded object downloads. This is why choosing one fixed threshold is hard. Notice, however, that the gap between two adjacent requests is more constant and predictable, compared to the time differences between the original page and the embedded objects. Hence, we propose the use of the following *refined timing heuristic*: For each referer page R, maintain time T_A as the "latest activity time". The activity can be either the last response packet being received for this referer page (T_R), or be a GET request sent for an embedded URL of this referer page (T_G). When a new request is sent with referer R at time $T_{G'}$, we check if $T_{G'} - T_A < T_{th}$, where T_{th} is a chosen threshold. If the condition holds, we classify this request as an embedded URL for R and also update $T_A = T_{G'}$.

The think time distribution of the refined timing heuristic is shown in Figure 2. Most adjacent requests (almost 90%) are within about 100-500ms, which is much less than the human think time. Of course, if the chain of requests is long, the chances of false positives will increase since a user may click on some link. However, the chance of a user clicking on a link before the page completely loads is quite small and this approach therefore works for almost all practical cases. The problem with the naive timing heuristic was that it was trying to choose one threshold for all web sites, whereas the refined heuristic adapts to the number of objects and to the time taken to download all the previous objects.

Note that the tail still contains a small percentage of requests that were sent almost 1000s (about 16.67 minutes) after the previous request. While this may seem like a user click, our file-type heuristic indicates otherwise. On further investigation, we found that the browser was requesting the same embedded object several times (with the same referer). This happens every so often, as in an auto-refresh. If the browser refreshes certain objects automatically after a long time, it could be difficult for us to correctly classify these refreshes as embedded requests. However, for the refresh requests present in the trace, we observed that the file type was almost always of embedded variety. So, our file-type heuristic would have correctly flagged them as embedded.

D. Overall Algorithm

The cobble tree construction algorithm combines the multiple heuristics discussed above: CDN detection and embedded object detection based on both file-types and refined timing. To combine them, we run them as two separate procedures. In the first procedure, we detect the CDNs or main supporting domains for each target web site by using the CDN detection algorithm described in Section III-B. Note that the administrator can choose to include multiple CDNs or supporting domains here based on the few top domains flagged using the algorithm. In the second procedure, we use the detected CDN domains along with the file-type heuristic and the refined timing heuristic to classify the traffic. Given a list of targeted origin domains,

```
Algorithm:
for each request req
BEGIN
    if host(req) ∈ Origin
        root(url(req)) = Origin
    else if (host(req) ∈ CDN &
            root(referer(req)) ∈ Origin)
        root(url(req)) = Origin
    else if (url(req) is embedded file-type &
            root(referer(req)) ∈ Origin)
        root(url(req)) = Origin
    else if (req passes refined timing test &
            root(referer(req)) ∈ Origin)
        root(url(req)) = Origin
    else
        root(url(req)) = NULL
END
```

Fig. 4. Overall classification algorithm for one domain

the goal of the algorithm is to classify each connection either as belonging to one of the target domains or as NULL when it does not belong to any target domain.

We classify each HTTP session based on the request message that the client sends to the server. Figure 4 shows the procedure for classifying a request. Each URL is associated with a "root" domain, which can be either NULL or one of the target domains. During processing, the heuristics are applied according to the specified precedence. Note that although conceptually we are isolating the session tree for each target origin domain, we do not need to maintain one unified data structure for the entire tree. Instead we can just maintain the root for each URL so that we know which tree this URL belongs to.

The precedence rules in the algorithm in Figure 4 are intuitive. If the host belongs to the origin domain, or to the CDN domain provided that referer's root belongs to the origin domain, then the host belongs to the origin domain. Then, we perform the file-type and timing checks, coupled with whether the referer's root belongs to the origin domain. Thus, if there is a false positive in the timing heuristic (*i.e.*, a GET request was sent to a third party host as a result of a user click and was not automatically fetched by the browser, but was misclassified by the timing heuristic) the overall algorithm is robust enough to stop further misclassification. For example, while the CNN page is loading, suppose a user clicks on some third party link, say facebook.com, on the CNN page. Because the page is still getting loaded, this user click may be inadvertently classified as an embedded object by the timing heuristic. The algorithm will set the root(facebook.com) to the origin domain (CNN). Further accesses to links from this third party page, however, will not match any of the rules because their referer would be facebook.com. For this one false positive to cascade into including an entire browsing tree, the user must repeatedly click on one link after another while the page is loading, with virtually no think time, which is difficult.

The algorithm also handles URL shorteners flawlessly. An URL shortener redirects a short URL to the actual long URL using HTTP's 301 return code. When a browser fetches the long URL, the referer field remains NULL and so the cobble tree for the long URL can be formed in its entirety as if the redirect never happened. In a similar fashion near domain names (*e.g.*nyt.com and nytimes.com) can also be handled since usually the shorter name redirects the browser to the longer one. The algorithm can be made even more robust by filtering out the links to embedded objects by parsing the content within the HTTP response. This not only helps reduce the number of false classification of the URLs, but also remedies against auto-refresh and other fraudulent activities trying to circumvent the cobbling process. We however, chose not to implement that in our current algorithm due to the large overhead involved in storing, uncompressing and parsing the content within a HTTP response.

IV. Experimental Evaluation

Before we evaluate the accuracy of CobWeb system, we first describe our experimental setup.

A. Evaluation Methodology

Packet trace and ground truth We take 70 out of the top 100 US web sites listed in Alexa, along with the top 26 popular web sites for users in the campus network. We exclude the sites that heavily rely on HTTPS (*e.g.*, gmail.com) because our algorithm is not applicable there, and those that require logins since ground truth is hard to obtain. We have collected two packet traces, Field2011 (227GB compressed trace on July 29th, 2011) and Field2012 (512GB compresssed trace on Jan 19th, 2012), from a 10Gbps large university network gateway link. Obtaining ground truth from traces in the field is challenging since there is no easy way to distinguish between user clicks and embedded downloads—a key requirement for measuring false positives and false negatives. Parsing the base page does not work since modern web sites heavily rely on javascripts to generate URLs dynamically. Thus to establish credible ground truth, we also simulate a real user clicking on the 96 web sites and collect these traces. We use a web browser to open each of the main web pages and wait for 1 min to record the traffic trace for each session. Hence we obtain 96 manual traces one for each web service, totalling 108MB.

False negatives To evaluate false negatives, we mainly rely on manual downloads in a controlled environment. We run the algorithm on the manual traces and compute the *false negative rate*, the proportion of traffic that is in the manual tree but is missed by the cobbled tree returned by the algorithm. For convenience, we also use its complement *true positive rate* in our discussion.

False positives Unlike false negatives, it is difficult to evaluate false positives using manually downloaded web pages, because by construction, we are not injecting any interference in the user click emulation. We address this problem by combining campus trace with controlled browsing as follows. First, we use our cobble algorithm to generate the *cobbled tree* for each user browsing session of each target web site in the campus trace. Recall that the root of each session tree is the starting point for each browsing session for a web page. The leaves of the

CDN domain	Origin domain	From Main	From Self	Others Others
2mdn.net	doubleclick.net	0.247	0.149	0.604
gstatic	google	0.854	0.006	0.140
images-amazon	amazon	0.523	0.207	0.269
imgur	reddit	0.749	0.107	0.144
imwx	weather	0.373	0.263	0.364
mzstatic	apple	0.919	0.001	0.080
turner	cnn	0.428	0.215	0.357
twimg	twitter	0.489	0.011	0.500
yimg	yahoo	0.544	0.105	0.351

TABLE I
CDN TRAFFIC IN PROPORTION OF BYTES

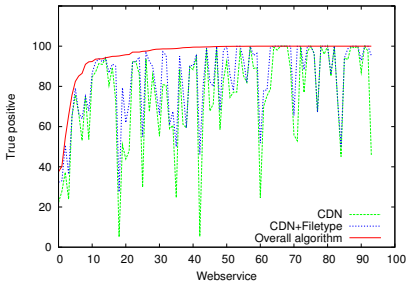
tree include both the embedded objects that are automatically fetched from any server, and user navigations to other pages within the same web site (as in Figure 1(b)). In order to find out the accuracy of this tree, we use the manual trace as the ground truth (we call this the *ground tree*) and compare it against the cobbled tree. However, one issue is that it is not easy to emulate the user opening other pages within the web site; for ease of evaluation, therefore, we only consider root page downloads (called *pruned cobbled tree*).

Ideally, both the ground tree and the pruned cobbled tree should overlap perfectly; branches missing in ground tree implies false positives. However, many web sites have very dynamic content which can cause the trees from two different sessions differ and introduce "noise". Such content is typically dynamic ads. For example, when we make two consecutive downloads for cnn.com, each download generates 150 and 148 GET requests, respectively. Among the 150 requests generated in the first download, 33 requests do not appear in the second download, with 25 being ads. Hence to address this problem, we mark a node as false positive if the following two conditions hold: (1) it is in the pruned cobbled tree, but not in the ground tree; and (2) its URL belongs to third-party non-ads domain. If a node is marked as false positive, then the entire sub-tree below the node is also marked as false positive.

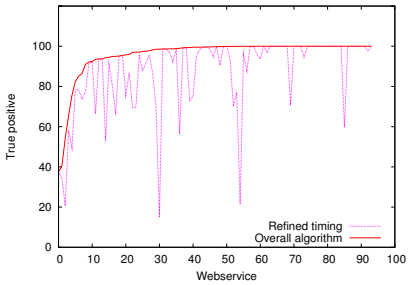
B. Results

CDN Detection In order to test the CDN detection heuristic, we first run the CDN detection algorithm with the Field2011 trace and then verify the detection result by using tools such as whois, dig and google search. Table I shows a sample of the detection result (for a subset of websites for space reasons). CDNs for most web sites are correctly detected except reddit.com, for which imgur.com was mistakenly identified as the CDN or main supporting domain. The reason was that lots of reddit users are browsing image links of imgur during the measurement period.

Table I also shows the traffic break-down for each CDN (or main supporting) domain. Traffic for each CDN domain can be referred by either the main domain, the CDN domain itself, or other domains. We observe that although in most cases the majority of traffic for a CDN is referred by its main domain, significant fractions of traffic for many CDNs are referred by external web sites. This confirms our intuition that classification just based on host domains is not sufficient. For



(a) True positives for CDN and file-type heuristics



(b) True positives for refined timing heuristic

Fig. 5. Proportion of traffic correctly classified by individual heuristics and overall algorithm. Webservices are sorted based on the True positive rate for the overall algorithm

example, even though 2mdn.net is the main supporting domain for doubleclick.net, majority of its traffic is referred by external web sites to show ads, and hence should be classified as part of the corresponding external web site traffic. The same can be said for twitter.com's supporting domain twimg.com.

We further run the CDN detection algorithm with the Field 2012 trace. For all 96 top sites, we find that the algorithm correctly identified CDNs for 89 sites (92.7% accuracy). In the other cases another third party site is marked as CDN incorrectly since there is no separate CDN domain for this web site and the third party domain exceeds the 5% threshold. There are also two web sites that use more than one CDN domain, so the algorithm just picks the top one.

Once the CDN or main supporting domain is determined for the target web site, we can apply the CDN heuristic to classify traffic. In order to find out the proportion of traffic classified by using the CDN heuristic, we construct the cobbled tree as follows: Starting from the target root domain with empty Referer field, we include a HTTP request into the tree if its host belongs to the root domain or CDN domain, and its referrer also belongs to the tree. Here we only evaluate the true positive rate since there are no false positives for the CDN heuristic. We use the 96 manual traces for this evaluation. Figure 5(a) (the green line) shows the proportion of traffic correctly identified by using the CDN heuristic. We observe that true positive rate varies across a wide range between 5.09% to 100%. The average true positive rate over the 96 sites is 75.68%. This indicates that the CDN heuristic is very effective for many web sites, but if used as the only heuristic, it is not robust enough to classify traffic for all web sites.

File-Type Heuristic We next investigate how much the file-

type heuristic can further improve the classification result. We construct the cobbled tree in a similar way as before, and in addition, include a session if the URI is for an embedded file type and its referrer host belongs to the tree. Similar to CDN heuristic, file-type heuristic does not generate false positives and we use the 96 manual traces for evaluation. Figure 5(a) (blue line) shows the proportion of traffic classified correctly by combining file-type heuristic with the CDN heuristic. Note that such embedded files can be downloaded from either CDN or third party. As a result, the two detection methods have certain overlap between their results. We observe the effectiveness of the file-type heuristic varies across web sites. On an average, the method with combined heuristics can correctly classify 83.75% of total traffic, an 8.08% increase over the original CDN heuristic. In other words, the file-type heuristic classifies an additional 8.08% traffic for embedded file downloads from third party websites.

Refined Timing Heuristic To evaluate effectiveness of the refined timing heuristic, we construct the cobbled tree using this heuristic, and use the method discussed before to evaluate mainly true positive rate. We find that the T_{thresh} value of 500ms gives a good tradeoff between false positive and false negative rates, and hence we use this value across our study. We evaluate true positive rate by using the 96 manual traces as before. Figure 5(b) shows the true positive for each target web site. We observe that the true positive rate varies across a wide range between 14.98% to 100%. The average true positive rate over all web sites is 87.11%. Again, as shown in the Figure 5 the combination of all these heuristics has the best chance of correctly classifying most of the traffic. We discuss this next.

Overall Algorithm We now evaluate the overall algorithm (shown in Figure 4) that combines all the heuristics. We first use the manual traces to evaluate false negatives, and then use both Field2012 campus trace and manual traces to evaluate false positives. In the latter case, there are typically multiple browsing sessions for each of the 96 sites. We take the average false positive rate of all the sessions for the same site. Figure 6 shows both the false negative and false positive rates by using the overall algorithm. Timing threshold is 500ms. Result is shown in two rows, sorted based on false negative rate. We observe that the overall algorithm performs very well for most web sites. The average false positive rate across all web sites is 3.54%. With very few exceptions, the false positive rate is below 10%. One anomaly is www.reddit.com with false positive 65.12%. We conjecture that this is caused by very frequent content update due to active user postings. Since we compare all sessions within the 5 hour field trace with just one ground truth trace, the "ground truth" is outdated for most sessions. To verify this, we collect a shorter 20 min trace on the same campus gateway along with a ground truth trace. For the 48 reddit sessions that we capture, the average false positive rate is now 5.2%, indeed much lower. This confirms that the high false positive rate we have observed in Figure 6 is indeed caused by the evaluation artifact rather than the cobbling algorithm.

The average false negative rate is 4.14% across all web services. Except the last 8 sites, all false negatives for all other sites are below 10%. Further investigation of the trace shows that the high false negatives for the last 8 sites are mostly caused by flash video players. Unlike the browser, the video players often have empty Referer field in their GET requests, hence our algorithm would not associate the request to the current session tree and miss the traffic completely. Not setting the Referer field is alright for the CDN or main domain , since they will still be accounted for, although as part of a different cobbled tree.

We think this problem can be addressed by correlating the sessions without Referer field with those with Referer field. For example, when we see a GET request with empty Referer field, we take its URL and find the "best matching" URL of another request among all the cobbled trees of the same user within a certain time period, say 5 sec. The intuition here is that if the flash player retrieves video from a domain, the browser is likely to retrieve some other content from the same domain. We have tested out this approach on the four domains with highest false negative rate, and found that their false negative rate is reduced to 19.1%. Work is still needed to further validate such approach, and exploring other alternative correlation methods.

V. Discussion

Mobile Apps. Mobile phone browsers use Referer field similar to the desktop browsers. However, our analysis using an Android phone indicates that mobile apps often do not set User Agent and Referer fields; CobWeb therefore cannot be applied for mobile apps. However, further investigation reveals that the mobile apps send requests to a significantly smaller number of domains; for randomly selected 17 popular web service apps, we found one average only 5 domains are accessed and 90% of traffic belongs to main and supporting domains (CDN) for the app. Hence simple static traffic rules may work well.

Video Services. Our algorithm does not directly apply to video service sites (*e.g.* Hulu, Netflix) that use HTTP streaming to deliver video content, since they typically have multiple domains for supporting content delivery. However, since there are very few of such large scale video service sites, it is affordable to make specific rules for such web services [3].

Disabling Referer. The cobbling algorithm can potentially be vulnerable to disabling Referer field. However, unless most users disable it, we can detect anomalous user behaviors and easily blacklist them. Collusion between users and providers makes the detection more difficult, but is rare since there is no clear incentive.

Prefetching and Auto-Refresh. Prefetching at the origin or CDN server does not affect our algorithm. In addition, prefetching by the client browser is similar to user accessing such content, so it will be classified accordingly. Auto-refresh is also fine, since it is similar to the user accessing the content, and hence billed accordingly.

NAT and multiple sessions. For network behind NAT with single external IP address, all the users inside the network would be correctly treated as one single giant user by the ISP for billing purposes. Multiple tabs inside a browser of a single

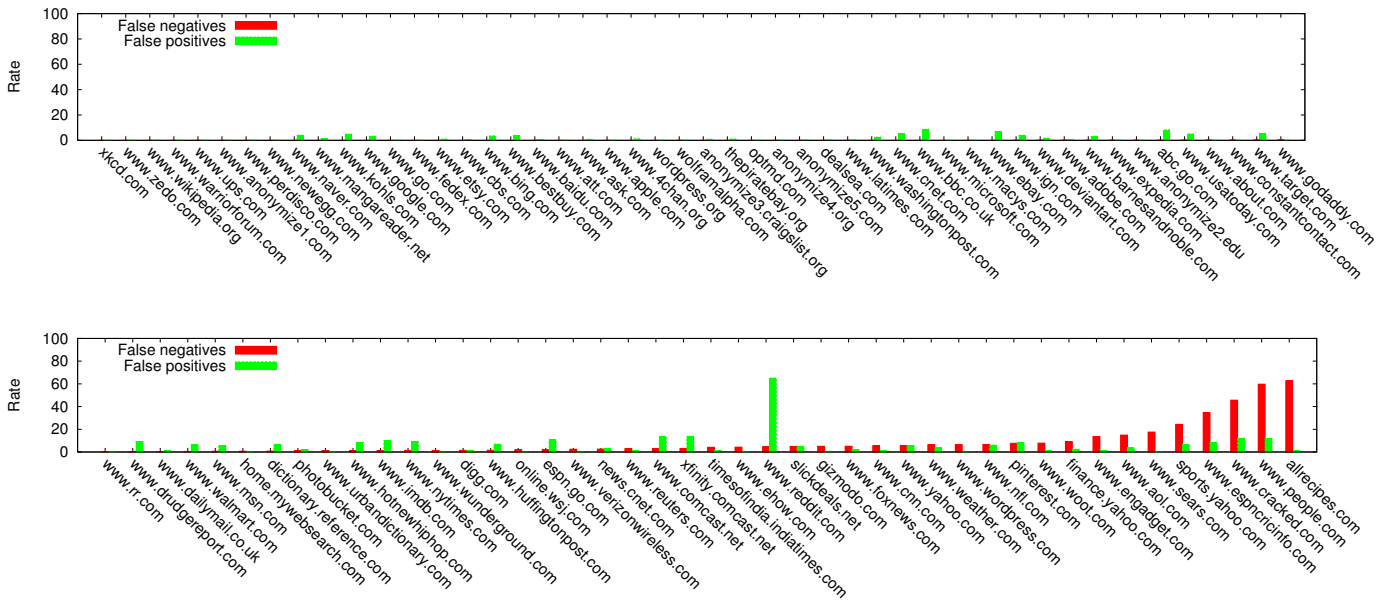


Fig. 6. False positive and false negative for all the webservices with timing threshold 500 ms.

user would not affect the accuracy of our algorithm because of chaining of requests by the Referer field.

VI. Related Work

Feldmann et al. uses a simple method for classifying traffic into web services by mapping hostnames to IP addresses [5]. Although such method worked several years ago, it is no longer suitable for the much more sophisticated web services today. For instance, if cnn webpage includes embedded objects from facebook, such objects cannot be associated with cnn based on IP addresses. In addition, same CDN server IP address may serve multiple web services.

In work by Butkiewicz et al. [4] they analyzed the complexity of modern web sites in terms of number, types and size of objects downloaded, and contribution of non-origin domains like CDN. This measurement study finds the impact of complexity on user experience. Research work has also been done in the past to catalog web services based on the type of their services, by using techniques such as machine learning [12], [8]. The focus in our paper is not to catalog the web services, but to isolate each target individual web service traffic to assist better billing and service management.

There exists related work in traffic classification and identification in general, especially at the application level [11], [13], [7]. There also have been some measurement studies to understand the new web technologies such as Ajax (*e.g.*, [14], [10], [9]). Our goal however is to provide a mechanism to identify sessions corresponding to a web service. Finally, a recent paper [6] proposes an algorithm called StreamStructure that also exploits the Referrer field for grouping web requests together, but the goal of that paper is to traffic characterization and not developing an online system for cobbling web traffic.

VII. Conclusions

We have presented the CobWeb system for in-network cobbling of traffic associated with a given set of web services. Such system can enable new types of monitoring and measurement capabilities, and can potentially enable new revenue models such as reverse billing for service providers. Our extensive evaluation suggests CobWeb can achieve low false positive and false negative rates. We view CobWeb as the first step towards solving the complex problem of web service classification.

References

[1] Allot. http://www.allot.com.

[2] Sandvine. http://www.sandvine.com.

[3] V. K. Adhikari, Y. Guo, F. Hao, M. Varvello, V. Hilt, M. Steiner, and Z.-L. Zhang. Unreeling netflix: Understanding and improving multi-cdn movie delivery. In *IEEE INFOCOM*, 2012.

[4] M. Butkiewicz, H. V. Madhyastha, and V. Sekar. Understanding website complexity: measurements, metrics, and implications. In *IMC*, 2011.

[5] A. Feldmann, N. Kammenhuber, O. Maennel, B. Maggs, R. D. Prisco, and R. Sundaram. A methodology for estimating interdomain web traffic demand. Oct. 2004.

[6] S. Ihm and V. S. Pai. Towards understanding modern web traffic. In *IMC*, Nov. 2011.

[7] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In *ACM SIGCOMM*, 2005.

[8] I. Katakis, G. Meditskos, G. Tsoumakas, N. Bassiliades, and I. Vlahavas. On the combination of textual and semantic descriptions for automated semantic web service classification. In *AIAI*, 2009.

[9] E. Kiciman and B. Livshits. Ajaxscope: a platform for remotely monitoring the client-side behavior of web 2.0 applications. *SIGOPS Operating System Review*, 41(6):17–30, 2007.

[10] M. Lee, R. R. Kompella, and S. Singh. Active measurement system for high-fidelity characterization of modern cloud applications. In *Proceedings of USENIX Conference on Web Applications*, 2010.

[11] J. Ma, K. Levchenko, C. Kreibich, S. Savage, and G. M. Voelker. Unexpected Means of Protocol Inference. pages 313–326, October 2006.

[12] N. Oldham, C. Thomas, A. Sheth, and K. Verma. Meteor-s web service annotation framework with machine learning classification. In *Int. Workshop on Semantic Web Services and Web Process Composition*, 2004.

[13] M. Roughan, S. Sen, O. Spatscheck, and N. Duffield. Class-of-Service Mapping for QoS: A Statistical Signature-based Approach to IP Traffic Classification. In *ACM IMC*, 2004.

[14] F. Schneider, S. Agarwal, T. Alpcan, and A. Feldmann. The New Web: Characterizing AJAX Traffic. In *International Conference on Passive and Active Network Measurement*, April 2008.