Abstract—The emergence of affordable mobile devices with rich interfaces and access to high bandwidth wireless connections has revolutionized mobile Web access. However, such new trends also imply downloading larger data volumes from the Web, with considerable battery and, often, monetary costs that inevitably degrade user experience.

One natural direction to tackle such a challenge is to exploit the substantial redundancy that Web contents exhibit. However, traditional techniques such as Web caching and data compression are unable to exploit a large fraction of the observable redundancy in Web traffic. Furthermore, more powerful techniques that have been recently proposed to support desktop Web clients fail to meet the specific requirements of mobile environments.

The mobile Web calls for end-to-end data deduplication that is able to achieve both high precision and minimal computational cost on the battery-constrained client side. We propose dedupHTTP, a novel deduplication solution that leverages the generic approach of Cache-Based Compaction to achieve the above requirements.

Using a full-fledged implementation of dedupHTTP with real workloads from popular Web sites, we obtained savings in traffic consumption of up to 94.5% when comparing to plain HTTP transfer, with minimal computational load on the battery-constrained client side.

I. INTRODUCTION

The Web has had a major growth in recent years. While the data available through the Web is doubling every year [1], the average Web page size and number of objects per page are steadily increasing [2]. Simultaneously, we resort to Web-based services for more and more tasks, most of which we used to perform offline. Examples range from Web-based mail, home banking, online news, document editing and sharing, to social networking.

The emergence of affordable mobile devices allows one to perform such web-based tasks anywhere, through smart phones, handheld and laptop PCs. While the resource limitations of older mobile devices forced content providers to deliver very simplified and limited alternative versions of their Web sites for mobile users to access, that is no longer the case. The fast growth of wireless bandwidth and the evolution of mobile device interfaces has revolutionized mobile Web access. Not surprisingly, mobile users now start to access increasingly richer Web content, downloading more and more information that they can now access and manipulate with acceptable time-to-display.

However, such new trends of the mobile Web come at a cost, which is often twofold. First, many wireless Internet providers charge a per-byte monetary cost. So, unless one is accessing the Web through a flat rate provider or a free hotspot, richer Web content will mean higher bills. Second, battery autonomy is not growing as fast as the number of bytes that the typical user is exchanging across power consuming wireless connections [3], [4], [5]. Inevitably, this considerably degrades the experience of mobile users, who are facing shorter device autonomy and, in many cases, paying higher access rates.

Hence, if we want to deliver such rich Web surfing experience to mobile users, we need to dramatically reduce the actual payload that mobile devices download across wireless links. One natural direction to tackle such a challenge is to exploit the substantial redundancy that Web contents exhibit [6]. In other words, to deduplicate the contents that mobile clients download from the Web.

Web deduplication is not a new problem, as well established techniques such as data compression [7] and classical Web caching [8] solve it to some extent. The former eliminates redundancy within the resource being downloaded, whereas the latter avoids downloading a resource that is entirely identical to a previous version present in the client’s cache.

However, there is strong evidence that a considerable amount of redundancy within downloaded web contents arises from other forms of redundancy, which escape both data compression and classical Web caching. The most common example is when a resource changes little by little over time, which is typically called resource modification [6].

Resource modification is prevalent in most page loads one makes every day. News sites have their main pages and articles updated several times per day, not only with fresh news updates but also with comments from their readers. A similar phenomenon occurs on social network pages, blogs, forums and discussion sites, just to mention a few examples. In all these cases, pages loaded at different times will often be very similar in content, with marginal changes introduced by the new pieces of information.

However, not all content is seemingly redundant. Usually,
image and video formats used on the Web (like GIF, JPG or Flash) are already compressed and do not exhibit much redundancy [6], [9]. Therefore, most Web deduplication approaches discard these resources from their targeted content [10], [11], [9], [12], a choice we decide to follow.

More recently, Rhea, Liang and Brewer have proposed Value-Based Web Caching (VBWC) [10], subsequently improved by Irmak and Suel [11], to address the Web deduplication problem. VBWC exploits all partial cross-resource and cross-version redundancy in the Web for clients with very low-bandwidth links (less than 80 Kbps). Their results show that, by detecting resource modification and other forms of redundancy that classical caching and data compression do not, they achieve substantial network consumption savings [10], [11].

VBWC's original motto is rapidly becoming uncommon as high-bandwidth links become ubiquitous across fixed and mobile Web clients. Still, one can consider VBWC outside the target population for which it was originally conceived. In particular, one can consider employing VBWC in the mobile scenario as a means of alleviating the battery consumption of increasingly energy-demanding clients. However, some characteristics of VBWC would be crucial limitations when deployed in the mobile Web scenario.

Firstly, VBWC relies on a dedicated middlebox proxy deployed at the client’s ISP, which intermediates all connections of the client with any Web server. While this assumption is acceptable in VBWC's target population of fixed clients, it may not always be a reasonable one for the mobile scenario. Mobile clients connect to distinct access points as they move through different geographic locations, which often cause them to roam across different ISPs. Assuming the client relies exclusively on a single proxy, maintained at a given ISP, would introduce undesirable additional inter-domain hops whenever the client is connected from a different ISP, and may even be prohibited if the proxy is behind strict firewalls. On the opposite extreme, assuming each ISP provides its own proxy would degrade VBWC’s effectiveness, as each proxy can no longer take into account the contents that the client caches through other ISP proxys.

A second disadvantage of the middlebox proxy-based approach is that it cannot exploit redundancy in page loads that are served through an HTTPS connection. The random noise that encryption introduces to the contents observed by the proxy conceals any redundancy that may be shared with cached contents at the client. Hence, any effort that the proxy carries out to detect redundancy will inevitably result in negligible success.

Finally, VBWC places most computational load on the proxy, but it still requires the client to maintain a hash table of every resource block it currently caches, as well as to perform look ups to such a table in order to decode the deduplicated contents it receives from the proxy. On devices with severe battery constraints, such client load has an energy cost that is expectedly far from being negligible.

In this paper we propose a novel deduplication system for the battery-constrained mobile Web, called dedupHTTP \(^1\). dedupHTTP eliminates the above limitations of VBWC when deployed in such a scenario, while retaining comparable deduplication effectiveness. The key insight behind our solution is that, by designing a deduplication solution that (i) is purely end-to-end [13] (without relying on any intermediate middlebox proxy) and that (ii) pushes all redundancy detection steps to the server, we can overcome the above limitations. Requirement (i) makes dedupHTTP well suited to roaming clients and allows it to work on plain text even when communication between client and server is encrypted, while requirement (ii) means that dedupHTTP places minimal computational load on the client.

With that in mind, we depart from the Cache-Based Compaction (CBC) approach proposed by Chan and Woo [14], which inherently ensures requirements (i) and (ii). Since the original CBC proposal is incomplete in its original formulation [14], as some crucial aspects are left unsolved, we propose non-trivial extensions to CBC that allow us to build a complete and practical solution.

More precisely, the main contributions of this paper are:

- We propose an efficient and scalable solution for the problem of allowing the server to track each client cache contents, a problem that was previously unsolved in CBC’s original work [14].
- We extend CBC to prevent malicious users from inducing CBC to disclose confidential data downloaded by other clients, a security vulnerability that is common to all trivial implementations of CBC.
- We build a full-fledged deduplication system for the mobile Web. This is, to the best of our knowledge, the first Web deduplication system based on the CBC approach to be fully implemented, deployed and evaluated. Results show that dedupHTTP achieves savings of network consumption of up to 94.5% when comparing to plain HTTP transfer.

The remainder of the paper is organized as follows. Section II describes dedupHTTP’s architecture, departing from the generic CBC approach and then detailing the improvements and modifications that we introduce to such a baseline. Section III proceeds to present the implemented system, which Section IV then evaluates. Section V surveys related work. Finally, Section VI draws some conclusions and describes future work.

II. ARCHITECTURE

dedupHTTP runs as two separate modules, one inside the client browser and another inside the Web server. The server maintains a set of resources, each one addressable at a given URL. The Web server can also create resources on-the-fly for certain requests. The resources stored by a server may evolve at any time it creates new versions of existing resources or new resources. For the sake of deduplication, whenever the server creates a new version of a resource that the server already

---

\(^1\) dedupHTTP is an open source project, available at http://deduphttp.sourceforge.net/
stores, it keeps the older versions at its local storage. As we show next, retaining older versions at the server increases the system’s ability to detect redundancy.

Hence, the server actually stores sets of one or more resource versions for each URL. For presentation simplicity, we refer to each resource version as a resource. Only when we need to distinguish between different versions addressed by the same URL we use the term resource version.

The client can obtain the contents of a resource stored at the server through HTTP requests (GET or POST). Upon receiving an HTTP request for a given URL, the server responds with the contents of the most recent resource version the server currently stores for such an URL.

The client stores a cache with the resources it downloaded from any server. Each resource cached at the client is indexed by the resource’s URL. For simplicity of presentation, we start by assuming that, for each URL domain, there exists one server only. Hence, from a cached resource’s URL, one may univocally determine the server from which the resource was downloaded. We discuss support for multiple servers per domain in Section VI.

At a glance, dedupHTTP follows the CBC approach. We now give a superficial overview of the generic CBC approach, before addressing detailed issues in the following sections.

CBC works in three main steps, involving a single network round-trip. On step A, along with an HTTP request for some target resource, the client tells the server which resources it currently caches. CBC’s original proposal does not explain how the client can perform this step correctly and efficiently. So, for the moment, we leave such an issue unsolved.

Upon receiving a request, the server proceeds to step B. It starts by locating the most recent version of target resource. Then, the server determines which resources it currently stores in common with the client’s cache. We call these resources the reference resources for the target resource.

The server then runs a local redundancy detection algorithm that determines which chunks of the target resource are identical to chunks in the reference resources. Clearly, such chunks of the target resource are redundant, hence they do not have to be transferred.

Finally, step C sends back to the client a response where the redundant chunks have been replaced by references to the reference resources. Since such reference resources are also cached by the client, it can directly obtain the redundant chunks from the local cache, instead of downloading them.

In the following sections we describe in detail how dedupHTTP performs each step.

A. Determining common resources

Each resource version is indexed by an unique identifier, assigned by the server on the first time any client downloads that resource version. Such an identifier is given by a (host domain, serial number) pair.

Whenever a resource is downloaded, the server appends the corresponding identifier to the response delivered to the client. The client stores that identifier along with each resource in cache. Each client uses the set of resource identifiers of its cache to tell the server which are the reference resources in cache at request time.

When the client makes a new request, it fetches all identifiers from the resources stored in the client’s cache that belong to the same host domain as the requested resource. These identifiers are serialized into a byte array and placed inside a new HTTP header called “Versions”, which goes on the request header.

For example, in Figure 1, the client’s cache has resources from two host domains, a.com and e.com. When the client makes a new request like "GET: a.com/3.html", the client introduces in the HTTP request header the identifiers of a.com’s reference resources that it currently holds in cache, i.e. 1.html and 2.html.

By filtering resource identifiers by host domain, we try to capture as much redundancy as possible from the least number of existing resources in the client’s cache. When a client sends a request to a server of a given domain, the client will only send the identifiers of: the previous version of the target resource (if cached), all other cached resources from the same folder and from the whole domain of the target resource. As Chan and Woo [14] observe, this mapping will not allow us to detect cross-domain redundancy, but the additional redundancy detected would not be significant, and the computational overhead would be noticeable.

The transfer of reference resource identifiers in a resource’s request from a client accomplishes two important goals. First, there is no need for the server to keep per-client state, as it is each client’s responsibility to provide the right list of reference resources. Secondly, it allows the client to use any arbitrary cache eviction policy, independently of other clients or server. The resources the client tells the server are shared between them, are the ones the server will use as reference resources.

Of course, we need to ensure that the resources identified on a request remain cached until the server response arrives. If, meanwhile, some of such resources were evicted from the client’s cache, the server might respond with references to redundant chunks that are no longer locally available to the client. In order to ensure that such situation never occurs, the client locks the affected cached resources until the response arrives or the server request times out.
1) Resilience to malicious clients: The algorithm described so far raises a security issue. A malicious client could carefully tamper the set of reference identifiers that such a client sends to the server, forcing the server to leak confidential data about previous downloads from other clients.

We illustrate how with a short example. Assume that a malicious client, Trudy, knows that client Alice has just downloaded confidential data from the server. Furthermore, consider that Trudy is aware that such confidential data can be either "Yes" or "No". However, Trudy does not know which option Alice actually downloaded.

Trudy could induce the server to disclose such information by asking the server to download a resource whose contents are "Yes", while carefully sending a forged resource identifier that is identical to the resource identifier of the confidential resource that Alice just downloaded. Therefore, if the server's response includes denotes that the target resource shares identical chunks with Alice's confidential resource, then Trudy may infer that the latter is "Yes". Otherwise, it is "No".

It is easy to see that this vulnerability would be shared by any naive instantiation of the generic Cache-Based Compaction proposal where it is the client that tells the server what are the client's cache contents. A possible solution to this problem would be to have the server monitor which are the contents of each clients' cache. However, not only would this imply higher storage requirements on the server, as it would require complicating the protocol with the necessary procedures for synchronizing the server's view of the client caches [10].

We propose a simpler and more lightweight solution that neither introduces network round-trips nor client-side load. We employ Hash-based Message Authentication Codes (HMACs) [15] to prevent tampering of such identifiers. The key insight to our solution is that the resource identifiers that the client exchanges with the server are exclusively issued and consumed by the latter.

Before appending a new resource identifier to a resource that some client has legitimately asked to download, the server also appends a HMAC of the resource identifier. The HMAC is generated with a secret key that only the server knows. The client, in turn, saves the (resource identifier, HMAC) pairs it receives in its cache. In subsequent download requests, the client sends such pairs to the server, so that the server can validate their integrity before accepting them. In contrast to the previous example, Trudy can no longer forge resource identifiers of resources for which Trudy has no permission to download.

It should be noted that our solution introduces very lightweight load on the server side. More precisely, computing and validating each HMAC requires two calls to a cryptographic hash function only. Most importantly, our solution introduces no computational load on the battery-constrained client, which acts as a mere carrier of (resource identifier, HMAC) pairs.

Finally, we should add that our method assumes that the client-server channel is secure. Otherwise, the malicious client could simply eavesdrop resource identifier pairs received by other clients and then replay them. One can easily ensure channel security by standard means, such as HTTPS [16].

B. Server-side redundancy detection

When the server receives a request for some target resource, we locate and fetch such a resource. After that, we determine which resource identifiers included in the request correspond to resources that the server is still storing, thus selecting the set of reference resources. Finally, we run a local redundancy detection algorithm to find chunks in the target resource that are redundant across the reference resources.

The server-side redundancy detection step can rely on any local redundancy detection algorithm. In our case, we present a different algorithm than the one proposed originally for CBC, which enables detecting redundancy across larger sets of reference resources than CBC's original algorithm, while retaining acceptable performance. It also makes it so that we do not need to store the full resource contents on the server and lets us optimize the algorithm when several requests are made to the same resource, both of which CBC encoding prohibits. We should note, however, that CBC's algorithm or any other algorithm could be used.

1) Chunk division algorithm: We start by creating per-byte hashes of the resource content. These hashes represent smaller chunks of the resource. This is done by using the rolling hash function of Karp-Rabin fingerprinting [17]. Let \( c_i \) be the byte at index \( i \) of the resource stream and \( k \) be the length of the Karp-Rabin fingerprint window. Let \( b \) be a prime number as base. The hash of the Karp-Rabin fingerprint is given by:

\[
H(c_i \ldots c_{i+k}) = c_i * b^{k-1} + c_{i+1} * b^{k-2} + \ldots + c_{k-1} * b + c_k
\]
Since $b$ is a constant, the iterability of this function allows us to calculate the next byte’s hash, with some simple operations on the previous byte’s hash with the new byte:

$$H(c_{i+1}...c_{i+k+1}) = ((H(c_i...c_{i+k}) - c_i \times b^k) + c_{k+1}) \times b$$

Now we have to select the hashes that represent the resource. We could select all hashes, but that is unfeasible memory-wise since we have a hash per byte of content. So, we select certain hashes to be the boundaries of larger chunks of data.

For this step we chose the Winnowing [18] technique, since it has been experimentally proven to give the better redundancy detection [19]. We enforce a minimum and maximum chunk size, because these content based methods can create too big or too small chunks, compared to the desired size.

After selecting the chunk boundaries, we hash each larger chunk using a 64 bit MurmurHash [20]. This choice was driven by the fact that MurmurHash outperforms cryptographic hash functions such as MD5 or SHA1, while retaining a very low collision rate, as the author shows [20]. We do not need cryptographic security in our hashes, so using MD5 or SHA1 would be overkill. Nevertheless, any stronger cryptographic hash function can be used in dedupHTTP.

2) Detecting redundant chunks: The meta-information returned after applying the chunk division algorithm to the target resource (chunk hash, offset within the resource and length) is stored in a hash table indexed by the chunk’s hash and on a chunk array. The chunks’ hash table is used for fast look up of a certain chunk in the resource, and the chunk array is used to orderly iterate the resource’s chunks (Figure 2).

In our example, the server fetches 3.html, assigns a unique identifier to it, splits 3.html into chunks and stores their metadata, as Figure 1 depicts.

The server then retrieves the resources identifiers from the received “Versions” HTTP header.

The server iterates through the chunk array with all chunk hashes of the target resource. Then the server gets each reference resource’s chunk hash table from a resource pool indexed by the resource identifier. For each chunk hash, the server looks up the hash table of each reference resource for an identical hash. If some chunk is not found on the reference resources, the server also looks it up in the target resource’s hash table. This way, we not only detect chunks that are redundant across the resources at the client’s cache but also inside the target resource being sent to the client.

Recalling the example in Figure 1, the server finds the identifiers of 1.html and 2.html in the HTTP request header. Then it goes through the array of chunk metadata of 3.html. For each chunk of 3.html it looks that chunk up in 1.html and 2.html’s hash tables. If the server did not find the chunk, it looks it up in 3.html’s hash table to see if that chunk already exists in the target resource.

On the server we do not store resource content. We only need the metadata of the resource to look up on subsequent requests. This is a major difference from the work of Chan and Woo [14] and Spring and Wetherall [9]. It results in a much lower memory overhead per resource at the server, allowing to store a history of resource versions metadata to help in future redundancy detection sessions.

**Algorithm 1** Pseudocode of the hash look up step

```plaintext
{the resource content is divided in chunks at resource creation}

resource ← newResource(resp.getContent())
storedResources.add(resource)
referenceIDs ← request.getHeader("Versions")
metadata ← newMetadata()
content ← newContent()
for all chunkHash in resource do
    foundChunk ← false
    for all refID in referenceIDs do
        referenceRes ← storedResources.get(refID)
        chunk ← referenceRes.getChunk(chunkHash)
        if chunk ≠ null then
            metadata.append(chunk.getMetadata())
            foundChunk ← true
            break
    end if
end for
if foundChunk = false then
    content.append(resource.getContent(chunkHash))
end if
resp.addHeader("Metadata", metadata.size)
codedResp ← resource.ID + metadata + content
resp.setHeader("Content-Length", codedResp.size)
resp.setContent(codedResp)
```

3) Algorithmic complexity: The algorithmic complexity of the server hash look up step is $O(n \times m \times p)$ in the worst case, when none of the target resource’s chunks exists on any of the reference resources (the pseudocode of this step is in Algorithm 1). $n$ is the number of chunks that compose the resource being served and $m$ is the number of reference resources from the client (including the current resource). $p$ is the number of chunks in the reference resource being looked up and is implied by the chunk hash table’s look up, whose complexity is $O(p/k)$, where $k$ the hash table’s array size. $n$ and $p$ are determined by the chosen chunk size and the total size of the corresponding resource, respectively. $m$ will be fairly small for an end-user client implementation (a user has few resources in cache for each domain). For small chunk sizes and bigger files $n \times p >> m$, so chunk size is the most relevant parameter.

4) Multi-request caching optimization: A resource can be requested by the same or different clients several times. On the first request the algorithm will be the one explained in the previous section. For subsequent client requests we can save an important step of processing, the division of the resource in chunks.

What the server does is map the served request’s ETag to
the respective resource metadata. These are unique resource identifiers present in the HTTP specification, created by the Web server. When a new request is to be served we check if the ETag has already passed on the server before. If it has, we retrieve the corresponding resource chunk metadata from server cache and continue to the response encoding phase. This way only the first request to each resource has to go through the chunk division phase.

C. Sending the final response

The final response begins with a metadata section. The size of this section (in bytes) is stored on a new HTTP header called "Metadata". This section’s content starts with the response’s resource identifier (black piece in Figure 1). Then the metadata section includes all the information the client needs to find the redundant chunks in its cache (grey pieces in Figure 1). Each chunk found on a reference resource is here identified by a four part tuple: the offset in the current response where the chunk is to be appended, the resource identifier of the reference resource where the client can find the chunk content, the offset in the reference resource where the chunk content can be found and the length of the chunk. The tuples are arranged in the same order as the corresponding chunks appear in the original response. At the end of the metadata block we append the non-redundant response content chunks, in the same order as they were present in the original response (white pieces in Figure 1). By respecting the order of the tuples and non-redundant chunks, the server can put them all contiguously.

In our example, the server will put 3.html’s unique identifier into the metadata section of the HTTP response. For each chunk found in any of the look ups to 1.html, 2.html and 3.html, the corresponding metadata is stored on the metadata section of the response. When all necessary metadata is included in the response, the server writes the size of the metadata section in the HTTP response header "Metadata". Then the server copies the non-redundant 3.html chunks to the end of the HTTP response.

When the client receives the response it only has to go through the metadata, copy the chunks referenced by the tuples to the final response, from the locally cached resources, and copy the non-redundant content from the received response to the final response, in the corresponding order. This way, the new resource is easily reconstructed on the client. The client does not even have to store meta information about any chunks. Since the metadata components are just resource offsets and chunk length, no hashes need to be sent, stored or looked up on the client. Only the resource identifier and the resource itself are stored on the client, in a table for the resource’s host domain that maps the resource identifier to the resource content.

In the example of Figure 1, the client receives the response and starts going through the metadata section. For each piece of chunk metadata, the client copies to the final resource the referenced chunk. This chunk comes from the corresponding reference resource, either 1.html or 2.html, or from a previously decoded part of 3.html. When a non-redundant chunk is needed, it is copied from the HTTP response to the final resource and the client continues to go through all metadata of the response. When the end of the metadata section is reached, the client copies the remaining non-redundant content to the final response. Finally, the client adds 3.html’s resource identifier to a.com’s array of identifiers, and maps it to the resource content.

We have opted to store the resource identifier in 2 bytes only, which helps shorten the metadata block, since each chunk reference has a resource identifier indicating where the chunk is present. This means there can only be 64K different resources referenced in the server. We believe this is more than enough for a regular origin server. Even with dynamic resources generated on-the-fly, after some days the older resources metadata can be evicted. Even if the client’s cache could accommodate for resources that were older than that, the usefulness of such resources is expectably low. Redundancy with fresh downloaded contents should mostly come from recent versions, rather than older ones, which we confirm empirically in Section IV-C.

The three other pieces of response metadata are stored in 4 byte integers. We have not considered lowering this size without further optimizations because it is easy to have an offset of a resource within the 4 byte range. The length piece can also achieve that range because of the metadata coalescing optimization. This makes for a 14 bytes total of metadata to reference a single chunk in the current implementation.

1) Metadata coalescing optimization: We have observed that there are consecutive redundant chunks detected in the same reference resource many times. Since we check the client’s resources in the same order for every chunk, we can save on metadata by considering a sequence of consecutive chunks as a single chunk. Hence, we only send a metadata block for the large coalesced chunk, instead of one block per redundant chunk (Figure 3). This optimization has a similar goal to hierarchical substring caching [11], but achieves better results in our system because we are not limited by the hierarchy depth.

A particular case where such an optimization is very effective is when a requested resource is aliased in the domain. The only thing that will be sent to the client is a chunk’s worth of metadata with the whole resource length.

This optimization has made it so that we use only 3 bytes on average to reference each redundant chunk, instead of the whole 14 bytes per chunk reference.

---

**Fig. 3.** Example of metadata coalescing on two contiguous chunks

<table>
<thead>
<tr>
<th>pasteOffset</th>
<th>resourceID</th>
<th>copyOffset</th>
<th>length</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>66</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1066</td>
<td>64</td>
</tr>
<tr>
<td>100</td>
<td>100</td>
<td>1000</td>
<td>130</td>
</tr>
</tbody>
</table>
III. IMPLEMENTATION

We deployed dedupHTTP’s client in a Web proxy to run at the Web browser machine and the server in a reverse proxy to run at the Web server machine. Both proxies were implemented using the OWASP Proxy [21] library, which takes care of every aspect in HTTP communication. Each proxy was extended with the proper functionalities described in the algorithms and architecture to deal with the resource content. This was done in about 500 lines of Java code, since that was the language of OWASP Proxy. The MurmurHash code was used from a public library [22] that ported the hash code to Java.

Using proxies for the implementation allows us to be browser and server independent, being easier to implement by not having to familiarize with a unique browser and server implementation, nor favor one over another. However, the results we intend to obtain in section IV are independent of the option taken in this regard. It has the inconvenient of adding proxy latency to the connection and having its own cache on the client. The proxy cache and browser cache will certainly overlap in most of their data.

Since the chunk size varies with the content (the expected size is just a median for reference), we bounded it from \( \text{CHUNKSIZE}/4 \) to \( \text{CHUNKSIZE} \times 2 \), so for e.g. a 64 byte expected chunk size the minimum chunk size is 16 bytes and the maximum is 128 bytes.

The resource metadata is stored per chunk on the server. Each chunk’s metadata is composed of two 32 bit integers, offset in the resource content and chunk length, and a 64 bit long for the chunk hash, resulting in 16 bytes of storage per chunk. The chunk’s metadata is stored on a hash table for the corresponding resource, indexed by the chunk’s hash. The chunk’s metadata is also referenced from an array inside the resource, which allows us to go through the resource’s chunks orderly.

In the server, the resources’ metadata is stored in an array of size 65536, whose index fits into the 2-byte metadata for the resource ID. In the client, each host domain has a hash table, which stores that domain’s resources, indexed by the resource identifiers supplied by the corresponding server. The client has a main hash table mapping all of the host domain’s hash tables to their host domain name.

We also implemented a delta-encoding algorithm to test against dedupHTTP. We store each requested resource on both the client and server. When a request is made for a new version of a resource the client has, we encode a delta between those two versions at the server. Then we send only the delta to the client, who reconstructs the new resource with the old resource version and the delta. We used xdelta3 [23] as the encoding/decoding algorithm.

For the Gzip and Hybrid algorithms we used the GzipInputStream implementation from OWASP Proxy, which inherits from the native GZIPInputStream of Java.

IV. EVALUATION

With our evaluation of dedupHTTP we aim to answer three important questions: how much traffic does dedupHTTP save for the client? For how much time is it useful to store resource version’s metadata on the server? Does dedupHTTP degrade Time To Display for the user?

The base experiment for the evaluation had a computer doing the job of client and another performing as server. The server has the resources of all workloads’ versions. The client has a list of the paths to the resources to test with for each experiment. In all experiments this list consists of resources from the first version of a workload (V0) and from a latter version (V1, usually the second version, except for the over time experiments). The client requests the resources from that list one by one (Figure 4).

The workloads for our experiments were created by downloading every news and comments page linked from the main pages of three worldwide popular sites: www.cnn.com, www.engadget.com and www.huffingtonpost.com (Table I). These pages are good representatives of dynamic content sites. We discard all image resources from the workloads, leaving only plain text, html, xml and javascript resources.

We downloaded these resources on two consecutive days at the same hour so that we could compare the redundancy retained from one day to the other, mimicking a regular user that likes to read the news once every day. For the CNN workload we also downloaded seven more days of content so we could study the evolution of results over a longer period of time. We removed unchanged pages from the second day, since they would be taken care of by the browser cache.

It’s important to note that the portion of resources prone to delta-encoding, i.e. resources with a previous version in the client’s cache, is 31.5% of the CNN workload, 17.3% and 19.7% of the Engadget and Huffington Post workloads, respectively. These are the only resources that delta-encoding acts upon and are the ones where dedupHTTP gets the best redundancy detection results.

<table>
<thead>
<tr>
<th>Workload Name</th>
<th>Number of Files</th>
<th>Total Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>CNN</td>
<td>337</td>
<td>42 MB</td>
</tr>
<tr>
<td>Engadget</td>
<td>315</td>
<td>36 MB</td>
</tr>
<tr>
<td>Huffington Post</td>
<td>401</td>
<td>62 MB</td>
</tr>
</tbody>
</table>

Table I: Two days workloads specification

![Fig. 4. Schematic of the base experiment](image-url)
TABLE II
KARP-RABIN FINGERPRINT SIZE VARIATION FOR THE CNN WORKLOAD

<table>
<thead>
<tr>
<th>Size of Karp-Rabin fingerprint</th>
<th>Redundancy detected</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>81.8%</td>
</tr>
<tr>
<td>32</td>
<td>80.6%</td>
</tr>
<tr>
<td>48</td>
<td>80.7%</td>
</tr>
</tbody>
</table>

All results here shown are the average of downloading a workload’s resources. The techniques we compared dedupHTTP to were delta-encoding, plain HTTP transfer with and without Gzip encoding enabled and an hybrid Gzip/dedupHTTP algorithm.

Since Gzip is very good at detecting redundancy on the resource itself, and dedupHTTP takes advantage at detecting redundancy with other reference resources, it only made sense to combine both techniques into an hybrid. We use dedupHTTP first because it is a content related technique, and using Gzip first would completely change that content. Then we apply Gzip to the much smaller dedupHTTP response, including the metadata. From now on we will refer to dedupHTTP with compression as "hybrid".

A. Minimizing transferred volume of data

DedupHTTP’s chunk division algorithm has two parameters that should be experimented for the best redundancy detection: the Karp-Rabin window fingerprint length (Table II) and the expected chunk length (Figure 5).

We have experimented the influence on redundancy detected of the Karp-Rabin fingerprint window sizes of 16, 32 and 48 bytes. 16 byte fingerprints give a better redundancy detection as shown in Table II, although by a small margin. This parameter has no influence in the number of chunks nor in the complexity of the division algorithm so we decided to use the one with best results (16 byte) for the remainder of the experiments.

We evaluated different expected chunk sizes, ranging from 32 bytes up to 8192 bytes (8 KB). Figure 5 presents our results. The smaller the chunk size, the more metadata the server has to store, since we have more chunks per resource.

There is a threshold below which the attained precision does not compensate the increased metadata overhead. The metadata associated with each divided chunk consumes 16 bytes of the server’s memory. Additionally, whenever that chunk is found to be redundant in a response to some client’s request, that chunk is replaced by a 14-byte reference. If the chunk size allowed for chunks smaller than these sizes we would have a reference to a chunk bigger than the chunk itself.

This is the case when we have a 32 byte chunk size, since there can be chunks down to 8 bytes, as described in section III. Clearly the lower bound for the expected chunk size is 64 bytes, because then we will have a minimum size of 16 bytes for the chunk size, which is larger than the communication overhead.

The other observation we can make here is that, while the server memory overhead increases roughly by a factor of two, the redundancy detected increases much slower. This may indicate that even though we get most redundancy detected with 64 byte chunks, a more conservative size may be better suited for a good redundancy detected/memory overhead relation.2

We also tested our hybrid algorithm with chunk sizes ranging from 64-byte to 1024-byte (1 KB) (Figure III). The redundancy detected with the hybrid is better for 128-byte and 256-byte chunk sizes, because of fewer metadata while still detecting most of the redundant data from other resources.

B. Overall transferred volume results

We selected dedupHTTP with 64-byte chunks and hybrid with 256-byte chunks to represent those techniques against the competition, since they yielded the best results.

When comparing all of the techniques redundancy detection (Figure 6), delta-encoding yields the worse redundancy detection. 2

we must not forget that a server may be shared between many clients, each requesting different resources increasing the memory overhead.
Gzip compression values are very consistent over all resources and workloads. For the CNN workload, dedupHTTP is better than Gzip (improvement of 7%), but not for the other two workloads (Gzip outperforms dedupHTTP by 25% and 9%, for the Engadget and Huffington Post workloads, respectively). The major reason for this difference is that Huffington Post and Engadget are very dynamic sites. News are added and removed very frequently from the main page. Each news item gets lots of new comments every day. With CNN this does not happen so rapidly. There is also more repeated code in the CNN pages than on the other sites, which makes it less dynamic.

Besides, we witnessed the number of resources prone to delta-encoding is bigger on the CNN workload. These are the resources where most redundancy will be found with dedupHTTP, since there is a previous version of the resource in cache from which much redundant content can be taken. Most of the other resources are just getting the layout, menus and some redundant text from sibling resources.

Finally, the hybrid algorithm surpasses all of the other techniques, with 6% to 19% more redundancy detected than Gzip, which should indicate the amount of data retrieved from resources other than the one being requested.

C. Redundancy variation over time

DedupHTTP stores older resource versions’ metadata on the server. This metadata occupies memory and has limited usefulness. As time goes by, people are less inclined to revisit a certain page, and even if they do, the page may have changed so much that the older content is no longer redundant.

So how long until the older version becomes useless? In order to know that, we experimented on the CNN workload for several days (Table IV). For each test day we downloaded the first day’s resources and the test day’s resources. We also compared them to the first day’s resource only. As we expected, there is more redundancy detected with the second day’s resources than with the first day’s resources only. As we can see, storing previous resource versions adds value to the redundancy detected by dedupHTTP (75.2% on day 1 and 81.9% on day 2). Then it decreases on the following two days and stabilizes on a value that is considerably lower (81.4% on day 3 to 79.2% on day 4).

This indicates that metadata should be stored in the server for just two days after it was served to a client. After that, the redundancy detected stabilizes and the memory overhead may become too much for the server to handle.

D. Results of Time To Display over the Internet

So how does the amount of redundancy detected and the overall complexity of dedupHTTP translate into the Time To Display over the Internet?

As we can see, the best redundancy detection configuration does not translate directly into the best TTD over the Internet for dedupHTTP (Table V). The impact on TTD over the Internet, of the slight decrease in redundancy detected, is offset by the smaller local execution time with larger expected chunk sizes.

We took the results of dedupHTTP with 128-byte chunks and hybrid with 256-byte chunks (the best results) and compared them with the other techniques. The comparison of the studied techniques with plain HTTP transfer has the purpose of seeing if there is noticeable improvement in TTD over the Internet for the user (Figure 7).

The most important result of this experiment is that Gzip, dedupHTTP and their hybrid all improve the TTD of all workloads, thus confirming that deduplication techniques improve the user’s Web experience, and that dedupHTTP does not degrade TTD for the user.

The second thing that pops up from this experiment is that delta-encoding delays the responses significantly for most workloads (2% more TTD than plain HTTP transfer, for the CNN and Engadget workloads; on par for the Huffington Post workload). The complexity of the algorithm is a fundamental drawback, rendering delta-encoding unsuitable for online encoding.
processing.

When using dedupHTTP with compression, the redundancy detection improvement over Gzip becomes relevant enough that the hybrid has a better TTD over Internet than Gzip for two of the three workloads (improvements of 0.9% and 1.5% for the CNN and Huffington Post workloads, respectively; decline of 0.5% for the Engadget workload).

E. Effect of multi-request caching optimization

Because of the multi-request caching optimization, dedupHTTP and consequently hybrid, also get some improvement relatively to the TTD value of Gzip, on subsequent requests. We experimented downloading the CNN workload after it was already requested and the TTD over the Internet of dedupHTTP with 64-byte chunks decreased significantly. We measured a decrease from 4770 ms to 4711 ms, already better than the TTD just using Gzip (4747 ms). The hybrid should follow this trend.

V. RELATED WORK

Chunk fingerprinting has been widely used in distributed file systems [24] and other deduplication systems for the Web (e.g., [10]). For chunk boundary selection these systems used the scheme proposed by Manber [25], but as Anand et al. [19] have shown the Winnowing [18] fingerprint selection technique is more profitable.

Manber’s fingerprint selection scheme selects each fingerprint that complies with $H mod P = 0$ where $P$ is the expected chunk size. On average they expect to have a fingerprint selected in $P$ bytes as a chunk boundary. Winnowing differs by always selecting the smaller/greater fingerprint in each consecutive window of $P$ bytes.

There have been many systems proposed for Web deduplication, each with its own limitations. Spring and Wetherall [9] proposed a shared cache architecture where two caches store chunks of data in a cooperative way. When redundant content is identified by the transmitting cache only the corresponding chunks’ fingerprints are sent. Their solution has the strong assumption of tightly synchronized caches, not addressing how to ensure that synchronization in the face of communication faults or temporary partitions, for instance. Such an assumption is particularly impractical for a Web server and its mobile clients. Spring and Wetherall follow a protocol-independent approach, but their results suggest that only HTTP is useful to analyze, since it is the most used protocol and with the most redundancy too.

Anand et al. [19] have recently confirmed Spring and Wetherall’s results for an enterprise setting (on a university setting P2P traffic is fairly more relevant). They also suggest that an end-to-end approach is preferable to a middlebox one, since most users do not share the same surfing habits. Therefore, the gain of detecting cross-user redundancy would be little, when compared to the cost of detecting redundancy across more data.

Chan and Woo [14] proposed a general approach to Cache-Based Compaction. It showcases two main ideas: a selection algorithm to choose reference objects for redundancy detection, and an encoding/decoding algorithm that acts upon a new object using the selected reference objects. They proposed a dictionary-based solution for encoding/decoding. When there are no common resources across the client’s cache and the server storage, it would act as traditional data compression. While when there is only one common resource, it behaves similarly to delta encoding. Their original solution does not scale well to more than a few common resources (in particular, [14] considers a maximum of three resources, including the object being downloaded). In contrast, the server-side redundancy detection algorithm employed by dedupHTTP scales gracefully to higher numbers of common resources (as Section IV discusses). Chan and Woo’s original proposal of Cache-Based Compaction [14] is incomplete, as crucial aspects are left unsolved. For instance, they do not address the problem of expressing the client’s cache contents to the server in an efficient and secure way.

Banga et al. [12] developed an optimistic deltas system. In optimistic deltas, when a proxy server receives a request for a new resource, it immediately responds with an older cached version of the resource, while it forwards the request to the origin server. When the latest resource version arrives at the proxy, it creates a delta between the older version and the current version of that resource. Finally it sends the delta to the client so that he can reconstruct the new resource. In cases where the idle time between the request and the server’s response is not enough to allow a complete transfer of an older version, their solution is able to abort the latter and send the new version in a plain fashion.

Optimistic deltas are not an end-to-end approach, hence are not suitable for the expectable usage patterns of mobile clients, as we discuss in Section I. Besides, with today’s high bandwidth this kind of latency reduction would be inefficient, more often than not overlapping the transport of the older resource with the response carrying the new one. Nevertheless, in cases where mobile clients always connect through the same access point, optimistic deltas can easily be combined with dedupHTTP in an attempt to boost its efficiency.

Rhea et al. devised Value-Based Web Caching (VBWC) [10], a solution that provides deduplication for clients connected across low bandwidth links (less than 80Kbps). VBWC relies on a ISP proxy. VBWC’s proxy maintains, for each client, a loosely synchronized set of the hashes of the chunks that such client is caching. The proxy does not store the actual data, which ensures the scalability of VBWC, as long as chunk size remains at coarse levels (2KBytes in [10]).

When a server sends some resource to the proxy, as a response to a given client’s request, the proxy divides such response in chunks and checks if that client already holds each chunk. If so, it only transfers the chunk’s hash. The client, in turn, maintains a local chunk hash table that indexes every chunk the client currently caches. For each chunk hash the client receives from the proxy, the client looks up the local hash table to confirm whether that chunk is effectively cached. Since the proxy may have an outdated view of the current
contents of the client cache, this look up step is required for correctness of the protocol. Irmak and Suel [11] further improved VBWC by employing an hierarchical redundancy detection scheme, which enables substantially smaller chunks to be considered, while retaining acceptable storage and network overheads. Their proxy maintains a multi-resolution view of the client’s cache, which includes chunks of different sizes. They employ a simple cache eviction policy that ensures that such a view includes hashes of more fine-grained chunks that have been recently accessed, and hashes of very coarse-grained chunks of older data [11].

As discussed in Section I, both solutions suffer from important limitations when deployed in the mobile Web scenario. Namely, their reliance on an ISP proxy and look up load that both place on the battery-constrained client side.

Both CBC and VBWC are state-of-the-art solutions for detecting redundancy on resource modification and intra-domain aliasing targets. But there is also a fair amount of inter-domain aliasing, albeit lesser than the previous sources of redundancy. This can be detected by other techniques like Duplicate Transfer Detection (DTD) [26], which can complement the previous solutions, including our own. In DTD, the server creates a digest of the requested resource with MD5 or SHA1 and responds with the digest. The client checks if that digest is indexed in its cache; if so it sends a message telling the server not to send the resource; if it is not indexed then the server sends the full resource to the client. Besides inter-domain aliasing being one of the less relevant sources of redundancy, DTD also needs a second roundtrip to function properly.

Data deduplication is not an exclusive problem of the Web. This problem has received much attention from other distributed systems, namely distributed file systems. Several research and industrial distributed file systems (e.g. [27], [24], [28], [29], [30], [31], [32], [33], [34]) employ data deduplication, not only for reducing network consumption, but also for reducing local storage usage. Perhaps the most prominent deduplication approach employed in the file system context is compare-by-hash [27], [24], [28], [29], [30], [34].

However, compare-by-hash has been designed with the assumption of resourceful clients, placing heavy computation efforts on the client side. While compare-by-hash is able to substantially minimize network usage, and therefore the associated battery usage, it requires clients to perform extensive local hash look ups. Not only can such local computation consume enough energy that dilutes the overall energy benefits of compare-by-hash, as energy consumption increases as one employs higher precision variants of compare-by-hash (e.g. by reducing chunk sizes or by employing more intricate variants, such as [11]).

Furthermore, compare-by-hash complicates the data transfer protocol with additional round-trips (i), exchanged metadata (ii) and hash look ups (iii). These may not always compensate for the gains in transferred data volume; namely, if redundancy is low or none, or when, aiming for higher precision, one uses finer-granularity chunks [35], [24], [36]. Moreover, any known technique for improving the precision and efficiency of compare-by-hash [35], [37] increases at least one of items (i) to (iii).

Muthitacharoen et al. [24] were, to the best of our knowledge, the first to notice a potential security vulnerability in a deduplication protocols where a malicious client that sends carefully chosen metadata to the server can gain access to unauthorized data from other clients. While such a vulnerability shares some resemblance with the one we address and solve in Section II, the former arises in the context of compare-by-hash systems. In Mandagere et al.’s 2008 survey [38], the authors state that such vulnerability of compare-by-hash remains unsolved.

VI. CONCLUSIONS AND FUTURE WORK

The new trends of the rich-content mobile Web imply downloading larger data volumes from the Web than before. While wireless bandwidth is keeping the pace of such a revolution, owners of battery-constrained devices are experiencing shorter device autonomy and, often, paying higher bills when connected through access points that charge them per each transferred byte.

One natural direction to tackle such a challenge is to exploit the substantial redundancy that Web contents exhibit. While traditional techniques such as Web caching and data compression are unable to exploit a large fraction of the observable redundancy in Web traffic, more powerful techniques that have been recently proposed to support desktop Web clients fail to meet the new requirements of mobile environments. The mobile Web calls for end-to-end data deduplication that is able to achieve both high precision and minimal computational cost on the battery-constrained client side.

We propose dedupHTTP, a deduplication solution that leverages the generic approach of Cache-Based Compaction to achieve the above requirements. Using a full-fledged implementation of dedupHTTP with real workloads from popular Web sites, we obtained savings in network consumption of up to 94.5% when comparing to plain HTTP transfer.

While our work suggests that Cache-Based Compaction is an appropriate approach for mobile Web clients, dedupHTTP is only a first complete instantiation of such an approach. For future work, we plan to study and compare alternative solutions based on Cache-Based Compaction. For instance, using alternative strategies for selecting the subset of resource identifiers to send to the server when initiating a request, or different algorithms for the server-side redundancy detection step (including the algorithm proposed initially by Chan and Woo [14]).

Another direction for future work is to study lightweight support for cross-domain or cross-server deduplication. A representative scenario is when a given domain is served by multiple replicated servers, a dedupHTTP client that accesses one of the servers will receive resource identifiers that are not valid when the same client later contacts a different server replica. Hence, any redundancy that might exist across the resources cached from the first server and the resources to
download from the second server will be neglected, not only by dedupHTTP's current design but also by the original Cache-Based Compaction generic solution. Enabling cross-server or cross-domain redundancy exploitation is not trivial. A possible solution is to identify individual resources by their hash value, instead of by a unique identifier assigned by the server from which the content was downloaded. This way, the resource identifier issued by a given server would still be valid when interpreted by a different server.

REFERENCES