Siskin: Leveraging the Browser to Share Web Content in Disconnected Environments

Samuel Sudar
Google, Inc.
Seattle, WA
sudars@cs.uw.edu

Matt Welsh
Google, Inc.
Seattle, WA
mdw@mdw.la

Richard Anderson
University of Washington
Seattle, WA
anderson@cs.uw.edu

ABSTRACT

Schools in the developing world frequently do not have high bandwidth or reliable connections, limiting their access to web content. As a result, schools are increasingly turning to Offline Educational Resources (OERs), employing purpose-built local hardware to serve content. These approaches can be expensive and difficult to maintain in resource-constrained settings. We present Siskin, an alternative approach that leverages the ubiquity of web browsers to provide a distributed content access cache between user devices on the local network. We demonstrate that this system allows access to web pages offline by identifying the browser as a ubiquitous platform. We build and evaluate a prototype, showing that existing web protocols and infrastructure can be leveraged to create a powerful content cache over a local network.

ACM Reference Format:
Samuel Sudar, Matt Welsh, and Richard Anderson. 2018. Siskin: Leveraging the Browser to Share Web Content in Disconnected Environments. In *COMPASS '18: ACM SIGCAS Conference on Computing and Sustainable Societies (COMPASS), June 20–22, 2018, Menlo Park and San Jose, CA, USA.* ACM, New York, NY, USA, 7 pages. **https://doi.org/10.1145/3209811.3209820**

1 INTRODUCTION

The web has tremendous potential to enable education for users in emerging markets. With an increasing amount of free or open-use educational content becoming available online (Khan Academy, Wikipedia for Schools, etc.), schools in developing regions can bring vast quantities of human knowledge directly into the classroom. However, the web is built with the assumption of fast, free, and always-on connectivity. This presents a substantial challenge to schools in developing regions, which often have slow or intermittent Internet connectivity, such as a flaky dialup connection [1]. While modern web standards make it possible to develop sites that can be used offline (e.g., using ServiceWorkers to persist content in the browser), this approach does not scale to the vast quantity of legacy web content.

The conventional approach to working around this problem involves caching static snapshots of web content, typically on dedicated hardware [2, 12]. However, this approach poses substantial logistical and cost challenges for schools in emerging markets. A typical edge cache box costs upwards of several hundred dollars, which can be prohibitive, and still requires manual maintenance and updates to software and content [8]. Schoolteachers, especially in developing countries, are not system administrators and cannot be expected to maintain esoteric hardware and software to support classroom web use.

In spite of their cost and maintenance burden, static educational content served on dedicated devices and accessed from desktop and laptop computers—referred to as Offline Educational Resources (OERs)—has become increasingly popular in the developing world [1]. These solutions have been criticized for their cost, difficulty to maintain, and for the opacity of their content, which is typically fixed or requires significant technical expertise to update [8].

We set out to determine if it was technically feasible to provide OERs without any dedicated hardware, using only existing infrastructure. Ideally such a system would run on commodity hardware, require no additional infrastructure, need only minimal configuration, and allow content to be easily curated. Our key observation is that modern web browsers have the capability to support everything needed to enable automatic, distributed caching of web content that can be shared across multiple users on a LAN. While the basic idea is not new—distributed caches have been explored since the foundations of the web [7, 17, 22]—our work leverages three key insights that represent a practical, deployable solution for real-world users today.

First, conventional HTTP caching is inadequate for supporting true offline access to Web content. Given that most web pages (and many resources that they depend on) are uncacheable [11], and the degree to which websites use dynamic and fetched content (e.g., using AJAX), standard HTTP caches cannot guarantee that a given page will be fully cache-resident and hence usable by a user without an Internet connection. Second, separating the functionality and maintenance of the cache from the device and software used day-to-day implies that functionality will erode quickly. The caching solution should be integrated directly into the device and software (e.g., the browser) that the user interacts with, to ensure it does not introduce a single point of failure and does not suffer from neglect. And third, modern protocols (such as Multicast DNS (mDNS) and DNS Service Discovery (DNS-SD)) enable automatic discovery and sharing of resources across a LAN segment, leveraged in products such as the Chromecast media-streaming device and others. This can enable seamless sharing across multiple users without overhead for configuration and management.

In this paper we describe *Siskin*, an approach to distributed snapshot caching of web content fully integrated into the Chrome browser. (A demo of the prototype has been presented at a networking workshop [21].) To use Siskin, the user simply needs to install a Chrome

COMPASS '18, June 20–22, 2018, Menlo Park and San Jose, CA, USA

ACM ISBN 978-1-4503-5816-3/18/06.
https://doi.org/10.1145/3209811.3209820

App and Extension. Siskin allows users to save snapshots of web content that they browse locally, and it automatically shares those snapshots with other users on the LAN. While browsing, Siskin automatically discovers snapshots of pages hosted by other users on the network, and it retrieves those snapshots over the local network. Siskin is built using existing Chrome App and Extension APIs to perform web content snapshotting, storage, network discovery, and peer-to-peer content transfer.

Siskin is targeted at educational settings in the developing world that are interested in using OERs. Some research has shown promise for the positive impact of these devices on education by enabling access to things like Khan Academy [1]. We make no claim on the pedagogical benefits of OERs and are not investigating their value to education. Instead we observe that OERs are growing increasingly popular despite their cost and limitations, and we believe cheaper, more robust solutions are desirable. We are interested in the technical question of whether such a system can be built using only existing infrastructure.

We demonstrate a complete working prototype of Siskin and evaluate its performance. We show that Siskin is effective at caching snapshots of content and enabling other devices to discover and retrieve snapshots while browsing. This unlocks the potential for offline web content to be made available in classrooms with a minimum of software configuration overhead.

2 RELATED WORK

Early work on web access in resource-constrained settings investigated Delay Tolerant Networking (DTN), which focused on designing network protocols to support intermittently environments that can occur in the developing world [14]. DTN approaches frequently went hand-in-hand with kiosk systems, which were custom built workstations developed to be deployed and shared in such settings [10]. Together, DTN and kiosks can be criticized as stovepipe solutions—hardware and operating systems must be custom built, and existing Internet infrastructure must be modified to accommodate new protocols.

A more conventional practice on challenged networks is to install an HTTP cache. Efforts at increasing the efficiency of caching solutions have included making multiple caches cooperate [7, 17]. The C-LINK system performs cooperative caching by using a coordinating proxy to store resources using clients' local storage [13]. Wolman et al. found that collaborative caching has promise for small populations, but concluded that "the crucial problem that must be solved to improve Web performance is how to increase page cacheability" [22]. This stems from the fact that modern web pages are composed of dozens of HTTP resources (e.g. CSS stylesheets, images, and JavaScript files), and the majority of this content is uncacheable [11]. Even if those uncacheable resources are not article text, broken images or failed styling can make content hard to consume and can lower the value of content. More recent approaches acknowledge the challenges of caching dynamic content and attempt to understand resources themselves that might be dynamic, introducing a cloud-directed profiler that increases traditional cache hit rates on challenged networks [19].

If caching conventions are ignored and stale resources are served, as in [13] and [3], resource-level HTTP caching remains inadequate as it assumes that devices on a network are at least partially connected, even if over a challenged backhaul. By design a cache first checks locally, and in the event of a miss it goes to the network. Since caching occurs at the network level, there is no way to inform the user of a miss. If a request misses the cache, the user might like to know that the request to the origin server will fail or will take more time than they are willing to wait. Conventional cache models do not allow this.

A number of commercial solutions have created content access hubs that enable local network sharing and host OERs. These can generally be thought of as web servers that respond to requests on a LAN. The Intel CAP, C3 Critical Links, and eGranary projects are examples [2, 6, 12]. Rachel Offline and Khan Academy Lite are software solutions built on this type of platform [15, 18]. These devices can cost on the order of hundreds of dollars, and maintenance has been shown to be a significant burden in resource-constrained settings [8]. Our work exists as a complement to these efforts, demonstrating that similar functionality can be provided without standalone hardware and without introducing a single point of failure.

Our work is most similar to [3], which implements aggressive HTTP caching as a Firefox extension. That project does not coordinate between machines, preventing users from benefitting from peer caches on the network. It is also aimed at accelerating browsing behavior, e.g. through aggressively prefetching links on a page from the Internet, rather than on distributing content. For these reasons their system is not well-suited as a platform for OERs.

3 SYSTEM DESIGN

Siskin provides a seamless distributed cache of web page snapshots available to any device on the same LAN segment; this will typically be a single classroom or a goup of classrooms. Every peer in the Siskin network hosts web page snapshots, which can be discovered and fetched by other peers on the network, avoiding the need for fetching content from a slow or intermittent Internet connection. Siskin achieves this using a combination of local page snapshotting and caching; peer discovery using mDNS; peer-to-peer snapshot fetching using Web Real-Time Communication (WebRTC); and a Chrome App to provide the UI. In this section we provide details on Siskin's architecture and implementation.

Our prototype is implemented as a Chrome App and Extension combination that communicate via the App and Extension APIs. Both components are needed due to the security model of Apps and Extensions in Chrome. Extensions are able to interact with the user's web browsing experience, e.g. to save a page as MHTML or to inspect navigation requests. Apps are able to access the system directly, e.g. to write files to disk and access system sockets.

3.1 Snapshotting Content

The fundamental unit in Siskin is the rendered page. This creates a one to one mapping between a top-level URL and a resource resident in the cache. This approach effectively snapshots the rendered web page that results from a visit to a URL. Unfortunately there is not yet a standard for distributing web pages that perfectly recreates a connected experience. The most widespread support is for MHTML, which takes a snapshot of the DOM, inlining external resources like images. This has the benefit of being a single hermetic file, making

distribution simple, and of being well-supported by browsers. When a saved page is viewed, it is fetched from the peer, saved to disk, and displayed in a browser tab.

In our prototype, MHTML snapshots are saved manually by users. When visiting a page, clicking the Siskin Extension icon in the Chrome toolbar adds the page to the cache. The Extension obtains the MHTML of the page and passes it as a blob to the App, which writes it to disk. For security reasons, we elected to keep snapshotting a manual process. We discuss security implications in greater detail below.

Alternative content ingestion methods are possible. A set of pages could be crawled and saved as MHTML, snapshotting them, and then moved into the Siskin directory on a host machine. This versatility allows existing content from OERs, including Rachel Offline or eGranary, to be shared using Siskin.

Although widely supported, a notable shortcoming of MHTML is the fact that it is completely static. As such it does not support JavaScript execution and by does not natively allow video or media playback. For this reason responsive sites and web apps are not well-suited to MHTML. This is discussed in more detail in the Discussion Section. An alternative approach could ignore cache-control headers and simply cache all resources, as in [3] and [13]. We find the single file, hermetic nature of MHTML to be a preferable distribution mechanism. Resource-level caches are best suited to configurations where a cache sits between a machine and the origin server, allowing individual requests to be handled in flight. Saving pages as an MHTML resource simplifies distribution by allowing alternative configurations, including look-aside caching behavior where a user is informed before navigation that a local load will succeed. Employing MHTML also simplifies distribution in creating a mapping from a URL to a single file. It is explicit that MHTML is only a snapshot, not a stale page served from a cache.

Siskin does not try to support web app behavior that requires complex interactions with a server. Email applications, for example, are not handled. Siskin aims only at operations where a page can be displayed without needing to interact with a server after the time the page is saved.

3.2 Peer Discovery

A discovery component is necessary to find peers running Siskin on the local network. We accomplish this by employing mDNS and DNS-SD. These zero-configuration protocols are a standard solution to the problem of network service discovery; they are employed by many services, including Chromecast media streaming devices [4, 5]. mDNS uses multicast UDP to issue DNS queries to the local network, while DNS-SD specifies how to use DNS records as a hierarchical database for service discovery. Using both together, clients can discover peers running a service and resolve an IP address and port combination to connect to the service. Chrome Apps provide an API to issue and respond to UDP requests, making an implementation of mDNS and DNS-SD straight-forward.

Alternatives to mDNS and DNS-SD exist. Simple Service Discovery Protocol (SSDP) is another solution that relies on multicast addressing to query and advertise via the local network. In some use cases, seamless discovery is not required and the peer to peer process can be bootstrapped by directly sharing connection information. A teacher might write the IP address of their host machine on the board, for example, or a WebRTC offer could be shared directly between users. Distributed systems such as [20] communicate with peers without keeping IP and port information for each individual device. These have desirable properties for large systems, namely that contact information is only needed for $O(\log N)$ peers to communicate with content that could be stored on any of the N peers. This is not an ideal solution for Siskin, which seeks to provide full enumeration of peers as a desirable property. For example, peer enumeration allows students to identify the teacher's machine and to browse only the content hosted on that machine.

3.3 Content Discovery

Local content discovery takes one of two forms. In the first, a list of peers on the network is presented via the App UI. Upon selecting a peer, the list of pages saved by that peer is presented to the user much like listing the contents of a directory. In the second, regular browsing behavior is augmented to inform users of locally available content. This process is referred to as "cache coalescence". To accomplish this, peers disseminate Bloom filters, as in [7], representing the URLs they have cached locally. The App component of Siskin maintains this information, allowing instances to locally determine if a given link is available from a peer. Links on a page are annotated to show that they are available.

By providing both modes of content discovery, Siskin is able to support sparse cache occupancy, where a user might save only a single page, as well as dense cache occupancy, where a whole domain may have been snapshotted. If a particular user has saved part of Wikipedia, for instance, users can first discover an entry point into that content via the summary listing. After navigating to the page, link annotation allows the user to browse only local content by augmenting regular browsing behavior.

Siskin annotates locally available URLs by communicating between the Extension and App modules. The Extension includes a ContentScript—a standard Extension component—that runs on on every page the user browses. This ContentScript queries the page for anchor tags that include the `href` property. These URLs are passed to the App, which holds the cache state of peers in memory as an array of Bloom filters. Each URL is checked in memory to determine if it is likely available on the network (the use of Bloom filters make this a probabilistic operation). Matches are passed back to the ContentScript, where the anchor tags are annotated with a small cloud icon to indicate their availability on the local network. More efficient alternatives, including distributed indexing as in [20], could be layered on top of Siskin. However, as discussed Section 4, that is not necessary for the contexts we are considering.

3.4 Data Transfer Between Peers

Data must be transferred between peers to perform content discovery and to view the snapshots. Our prototype employs WebRTC as the transport mechanism. WebRTC is a protocol suite designed to enable peer to peer communication between browsers. We chose WebRTC because it is built into the browser, and because all communication over WebRTC connections is encrypted.

Negotiating a WebRTC connections requires a signaling step, which is an exchange of text blobs referred to as "offers". In our

prototype implementation, WebRTC offers are exchanged via an HTTP endpoint using an open source HTTP server [9] we bundle in the App. The initiating peer issues an HTTP PUT request, the body of which contains the offer. The peer receiving the request extracts the offer, generates a return offer, and includes this offer in the HTTP response. With offers exchanged, a WebRTC connection can be established. We layer a messaging protocol on top of the connection's data channels, and Siskin instances use these messages to communicate.

3.5 Security and Privacy

We identify three main areas relevant to security and privacy in Siskin: 1) confidentiality of shared content; 2) secure communication between peers; and 3) integrity of cached content.

The first area relates to the confidentiality of content. Our approach of snapshotting pages makes this problem non-trivial. If a user elects to snapshot their email client or social media wall, for instance, they run the risk of leaking potentially private information to others on the network. To minimize the chances of this occurring accidentally, we elected to make snapshotting pages require a manual action from the user. In our prototype, content is guaranteed not to be shared until hitting the extension icon and electing to save content, allowing users explicit control over what content is available. Additional defenses could include blacklisting social media sites and implementing access control, allowing users to snapshot content for only their own use or to only share it with particular users.

The second area relates to secure communication. Siskin should support both encrypted and authenticated communication. The WebRTC transport mechanism ensures that communication is encrypted, but it is not authenticated. In connected contexts, WebRTC authentication occurs during the signaling step by communicating through an identity provider or via an HTTPS connection. HTTPS cannot be relied on in the offline settings Siskin targets. However, we could perform authentication by exploring a protocol akin to Secure Simple Pairing (SSP), which is used to pair Bluetooth devices. SSP provides a way to both exchange an encryption key and authenticate that the exchange was not subject to a man in the middle attack. SSP includes a numeric comparison mode by which two users compare several numeric digits, checking for equality. This exploits physical proximity of two devices and is appropriate for Siskin, where users are expected to be on the same LAN. SSP has been shown to be secure [16]. After SSP is complete, private keys can be shared and future secure communication can occur using RSA, provided that keys remain confidential.

The final area, integrity of content, is not provided by our prototype. Snapshotting and sharing are performed by untrusted peers, meaning that integrity guarantees stemming from the use of HTTPS are lost. Snapshots could be tampered with, or sophisticated peers could falsely claim that a fabricated snapshot originated from a specific domain. Integrity guarantees could be maintained by adding a level of trust to peers and cryptographically signing snapshots. An alternative could be to use a third party service to generate and sign snapshots. These approaches would require additional infrastructure and are not things we currently support.

4 EVALUATION

In this section we perform a technical evaluation of our Siskin prototype. We evaluate the capabilities of the technology in a laboratory setting as well as analytically. Another evaluation scheme could include deploying Siskin in an educational setting and collecting usage metrics. We instead approached evaluation as determining whether or not our approach is technically feasible given likely infrastructure at rural schools that employ OERs. Future work would be required to study Siskin under real world usage.

Our evaluation shows that Siskin's architecture meets our design requirements and could be employed as an OER solution. All our experiments were conducted on Acer 14 Chromebooks on the network in the University's Computer Science department. The Acer 14 is a mid-level Chromebook available for approximately $300 USD. The models used in our evaluations had a 1.6 GHz Intel Celeron N3160 Quad-Core Processor, 4 GB of RAM, and were running Chrome OS 58.

Metrics that depend on the number of pages in a cache are calculated for 1,000 pages. This number was chosen because many of the most highly rated Rachel Offline modules, which serve as OERs, contain on the order of 1,000 pages [18]. Additionally, 1,000 is a reasonable upper limit on the number of pages a user might add manually to Siskin without an automated content ingestion mechanism.

4.1 Data Transfer Speeds

Data is exchanged between Siskin peers both to communicate operational data (e.g., cache state) as well as the saved pages themselves. Siskin relies on browser JavaScript APIs for storage and network communication. The key question is whether browser-based peer-to-peer data transfer—i.e., WebRTC—is fast enough to serve cached pages between peers. To measure transfer speeds, we installed Siskin on two Chromebooks, connected them to the department's 802.11a/g/n WiFi network, and measured the time required to transfer files via WebRTC.

We generated files of 1 kB, 10 kB, 100 kB, 1 MB, and 10 MB, and measured the time to transfer each file 100 times between two peers. According to [11], 55% of all page loads transfer a total of 0-2 MB, and 95% transfer less than 10 MB. Siskin deals with snapshots of rendered DOM rather than raw resources, but we expect the size of raw resources to be similar to the size of the rendered DOM.

File transfer was conducted using two scenarios: WebRTC and WebRTC including connection establishment. In WebRTC, connection establishment is referred to as signaling, and is required for first time or infrequent requests to a peer. Mean transfer times over 100 runs are shown in Figure 1. The results are within the time a user might expect to load a URL on the network. For each file size, the WebRTC with Signaling step is slower by 300-500 milliseconds. This is expected, because the peer must first establish a connection, interfacing with browser APIs and performing an additional round trip with the peer hosting the file.

Transfer time does not increase perfectly linearly with file size. For example, both the 1 kB and 10 kB files transfer in approximately 85 ms when reusing an existing connection. The 1 MB file, meanwhile takes approximately 450 ms while the 10 MB file takes approximately 3 seconds, not the 4.5 seconds that might be expected to transfer 10 times as many bytes. This can be attributed to two

main sources of overhead. First, in our implementation each transfer creates its own data channel on the existing peer connection, which is shared overhead independent of file size. Second, WebRTC data channels are buffered, and the browser can inform client code to pause sending if the buffer becomes full before it has been sent to a peer. This allows WebRTC to respond to network conditions with minimal intervention from the client.

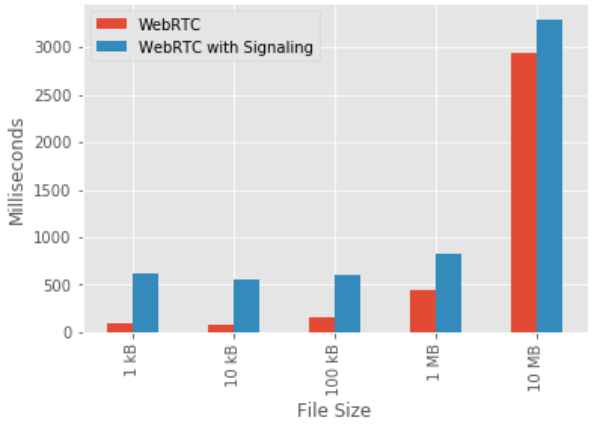


Figure 1: Mean transfer times for various file sizes between two peers.

4.2 Overhead for Cache Coalescence

Next, we evaluate the overhead of distributing cache directory information between peers in the network. This information allows peers to remain aware of what content is available on the local network and where it resides. As the number of peers and the size of the cache grows, the question is how much local network bandwidth is required to distribute this information. Our prototype of Siskin uses a fairly simplistic cache coalescence strategy—each peer exchanges a Bloom filter of locally-cached URLs with every other peer on a periodic basis (with a default of every 60 seconds). We perform this via unicast, where each peer sends a Bloom filter to every other peer. However, more efficient schemes, e.g., leveraging multicast, are also possible.

We calculate the theoretical load on a network with between 2 and 50 peers, each of which is caching 1,000 pages locally. Each peer encodes the list of cached URLs into a Bloom filter with a target false positive rate of 0.001, which requires 1,798 bytes. We consider an 802.11b network, as might be expected at a rural school, with an aggregate TCP throughput of 5.9 mbps. Maximum broadcast throughput is considered to be 0.5 mbps—one half of the lowest supported 802.11b rate of 1 mbps.

Under the fairly simplistic unicast system implemented in our prototype, fully distributing cache state requires each peer to communicate their state to every other peer. With 50 peers, this results in 4.4 MB (50 peers sending their 1.798 kB Bloom filter to 49 other peers) that must move across the network to fully distribute cache state. On the 802.11b network described above, this would require 6.0 seconds if all 50 peers joined the network at the same time. Figure 2 shows the theoretical bandwidth requirement of updating cache state via our unicast strategy under different refresh rates. This assumes that an initial distribution has been completed and the Siskin peers are periodically informing peers of their content by completely redistributing their Bloom filters.

With a 60 second refresh rate, the bandwidth impact is minimal. However, the unicast strategy is naive in that it requires an $O(N^2)$ operation in the number of peers, as each peer sends their directory information to every other peer. This can be improved by employing a broadcast mechanism to transmit the Bloom filters. Under this scenario each transmits a Bloom filter a single time, for a total of 0.09 MB. Given a broadcast throughput of 0.5 mbps, full distribution would take 1.4 seconds per distribution cycle if 50 peers joined at the same time. Figure 3 shows the theoretical bandwidth requirement of a broadcast coalescence strategy for different refresh rates. As with the previous calculation, this assumes that an initial distribution has been completed and the Siskin peers are periodically informing peers of their content. With a 60 second refresh rate, the bandwidth impact is minimal even with 50 peers.

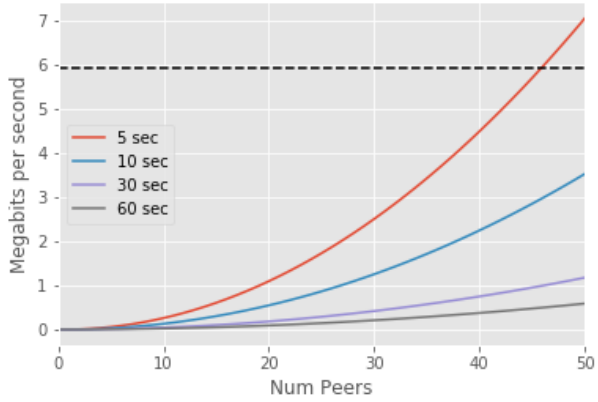


Figure 2: Bandwidth consumed by fully redistributing cache directory information via a unicast mechanism. Bandwidth is estimated at different refresh rates as additional peers join the network, each hosting 1,000 pages. The dashed line shows a theoretical maximum throughput of 5.9 mbps.

4.3 Query Latency

Finally, we measure the latency of querying the distributed cache state for a set of URLs and informing the user of the result. One of the key advantages of Siskin over conventional HTTP caching is greater transparency of available content. URLs entered into the address bar and URLs from anchor tags on loaded pages are queried against the body of locally available content. Siskin maintains in memory a representation of what content is available in the distributed cache. We use the term query latency to refer to the time it takes to query a set of URLs against this in-memory representation and update the UI to inform the user. (This representation is kept up to date with periodic updates, as described above.)

When a page is rendered, Siskin scans the HTML for all URLs contained in anchor tags in the page. It then queries its representation of the distributed cache state for the existence of each URL by

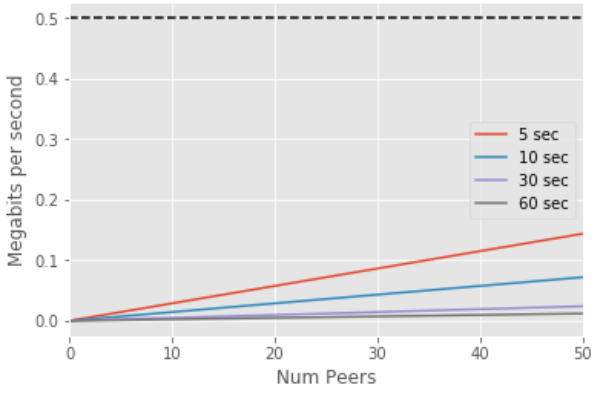
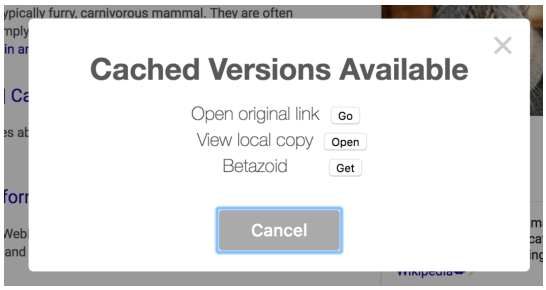


Figure 3: Bandwidth consumed by fully redistributing cache directory information via a broadcast mechanism. Bandwidth is estimated at different refresh rates as additional peers join the network, each hosting 1,000 pages. The dashed line shows a theoretical maximum broadcast throughput of 0.5 mbps.

(a) Outgoing links on cached pages, as well as on pages loaded from the Internet, are annotated to indicate if they are available on the local network. A cloud indicates that a peer has the linked page. A lightning bolt indicates that the linked page is available on the user's own machine.



(b) Clicking an annotated link creates a dialog that allows the user to choose how they wish to obtain the page. This can occur by going to the external network (if available), opening their own cached version, or getting a cached version from a peer.

Figure 4: The body of cached content is queried and exposed to the user during browsing.

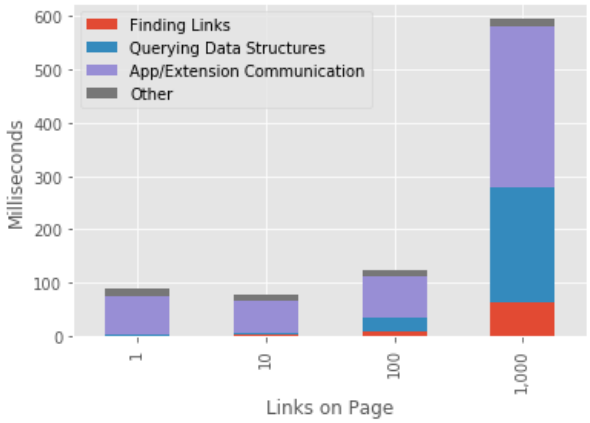


Figure 5: Mean times to query for URLs available on the network of 30 peers, each with 1,000 pages.

checking them against the Bloom filter it has received from each peer. When a page is found, Siskin annotates the URL on the page to indicate the cache residency status of each link. A lightning bolt indicates that it is available on the user's own machine, while a cloud indicates that it is available on a peer (Figure 4a). Clicking an annotated link creates a dialog that informs the user a cached version of the page available. The user can choose to try and load the original link from the Internet, view their own local copy, or get the cached page from a peer on the network (Figure 4b).

To measure query latency, we constructed four synthetic HTML pages with 1, 10, 100, and 1,000 outbound links. We also generated a synthetic cache directory containing 10 peers, each of which contained 1,000 pages. In this way the establishment of cache state was synthetic, but the querying on a local machine was measured. Each of the HTML pages was loaded 100 times. The cache directory was present in memory, as if peer state had already been communicated. The query latency is the time to look up all of the outbound links on the page in the cache directory.

The mean results over 100 runs are shown in Figure 5. Querying up to 100 links takes on the order of 100 ms, while querying for 1,000 links takes on the order of 600 ms. As shown in the Figure, the time it takes to find the links and query the data structures grows with the number of links, as can be expected. Communication between Chrome Apps and Extensions occurs via a messaging and callback system. Waiting for these messages to be delivered and callbacks to be invoked by the Chrome machinery constitutes the largest component of the query latency.

5 DISCUSSION

Siskin shows OERs do not require expensive, purpose-built hardware. It demonstrates that the web browser has become a sufficiently ubiquitous and capable platform that it can serve as a powerful offline content caching and distribution system. In educational contexts Siskin can complement existing efforts technologies that deliver OERs. We set out to investigate if OERs can be built using solely existing infrastructure. We are confident answering this in the affirmative without a deployment. Our approach demonstrates to creators of OERs that more flexible, affordable systems are possible. However, Siskin is not a perfect solution. In this section we outline limitations of the system, how those limitations could be mitigated, and what remains as future work.

The most immediate limitation is the capabilities of of MHTML as a distribution format. MHTML does not run scripts, meaning sites that depend on JavaScript after an initial render will not work as

expected. This does not impact all sites, and MHTML continues to be used in industry settings, but it is nevertheless a limitation of the format.

A more pressing problem is the fact that MHTML does not support video. Sites like Khan Academy are popular as offline educational resources [1]. Khan Academy lessons, as well as video services like YouTube, are not well served by MHTML. The DOM is saved but the media player is replaced by an empty HTML element. This could be mitigated by storing video files separately from MHTML and alerting the user to the associated files. This still would not be perfect, however, as the experience would differ from standard web browsing. A long term solution would be to develop a file format that better handles modern web content.

The current usage model of Siskin is based on URLs. Snapshotted URLs that are available can be listed, and links can be redirected to local copies. A useful feature would be the ability to perform keyword searches. Wikipedia becomes much more useful, for example, with the ability to search. Without search, users would have to rely on the directory listing feature of Siskin to find an entry point into cached content. Alternatively, a manual index page, like a site map, could be employed to explore content. Searching in an offline settings is complicated by the fact that content can join or leave the network with hosting machines. This suggests that machines should host their own search indices. Creating these indices in a way that serves real-world workloads while respecting the performance and storage capabilities of local hardware is left for future work.

The primary motivation behind Siskin is the idea that the ubiquity of web browsing technology can create distributed content distribution mechanisms. Our prototype, however, depends on Chrome App and Extension APIs. This means that it does not work on mobile devices, which do not support these APIs. An ideal Siskin implementation would run on any browser, not just laptops and desktops. Siskin is an example of how the browser can be used to replace stand-alone solutions. Our prototype demonstrates that this is feasible. We do not claim that our prototype is a complete solution. Instead we claim this demonstrates how the browser as a ubiquitous technology can expand the reach of web content even to those that are only intermittently connected.

6 CONCLUSION

We have presented Siskin, a system that supports OERs at schools in the developing world without requiring additional hardware or purpose-built devices. It achieves this by facilitating the distribution of web content on intermittently connected networks. Siskin leverages the ubiquity and capabilities of the web browser to distribute content between peers and integrate with the web browsing experience. Siskin represents the ability to replace or complement stand-alone solutions with existing, ubiquitous infrastructure—the browser—to give disconnected users access to the more than four billion pages that exist on the web.

REFERENCES

[1] Fundación Sergio Paiz Andrade. 2016. Evaluation Report: Assessing the use of technology and Khan Academy to improve educational outcomes in Sacatepéquez, Guatemala. (2016).

[2] C3 2017. Critical Links C3. **http://c3.critical-links.com/**. (2017).

[3] Jay Chen, David Hutchful, William Thies, and Lakshminarayanan Subramanian. 2011. Analyzing and Accelerating Web Access in a School in Peri-urban India. In *Proceedings of the 20th International Conference Companion on World Wide Web (WWW '11)*. ACM, New York, NY, USA, 443–452. **https://doi.org/10.1145/1963192.1963358**

[4] Stuart Cheshire and Marc Krochmal. 2013. *DNS-based service discovery*. Technical Report.

[5] Stuart Cheshire and Marc Krochmal. 2013. *Multicast DNS*. Technical Report.

[6] eGranary 2017. eGranary. **https://www.widernet.org/eGranary/**. (2017).

[7] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. 2000. Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol. *IEEE/ACM Trans. Netw.* 8, 3 (June 2000), 281–293. **https://doi.org/10.1109/90.851975**

[8] Nicci C. L. Gafinowitz. 2016. Digital Library Appropriation in the Context of SubSaharan Countries: The Case of eGranary Digital Library Implementation. In *Proceedings of the 7th Annual Symposium on Computing for Development (ACM DEV '16)*. ACM, New York, NY, USA, Article 29, 4 pages. **https://doi.org/10.1145/3001913.3006638**

[9] Github.com 2017. Web-Server-Chrome. **https://github.com/kzahel/web-server-chrome**. (2017).

[10] Shimin Guo, Mohammad Hossein Falaki, Earl A Oliver, S Ur Rahman, Aaditeshwar Seth, Matei A Zaharia, and Srinivasan Keshav. 2007. Very low-cost internet access using KioskNet. *ACM SIGCOMM Computer Communication Review* 37, 5 (2007), 95–100.

[11] httparchive.org 2017. HTTP Archive. **http://httparchive.org/**. (2017).

[12] Intel 2017. Intel CAP. **https://www-ssl.intel.com/content/www/us/en/education/products/content-access-point.html**. (2017).

[13] Sibren Isaacman and Margaret Martonosi. 2009. The C-LINK system for collaborative web usage: A real-world deployment in rural Nicaragua. In *Workshop on Networked Systems for Developing Regions*.

[14] Sushant Jain, Kevin Fall, and Rabin Patra. 2004. Routing in a Delay Tolerant Network. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '04)*. ACM, New York, NY, USA, 145–158. **https://doi.org/10.1145/1015467.1015484**

[15] learningequality.org 2017. Khan Academy Lite. **https://learningequality.org/ka-lite/**. (2017).

[16] Andrew Y Lindell. 2009. Comparison-based key exchange and the security of the numeric comparison mode in Bluetooth v2. 1. In *Cryptographers' Track at the RSA Conference*. Springer, 66–83.

[17] Radhika Malpani, Jay Lorch, and David Berger. 1995. Making World Wide Web caching servers cooperate, In Proceedings of the Fourth International World Wide Web Conference. 107–117. **https://www.microsoft.com/en-us/research/publication/making-world-wide-web-caching-servers-cooperate/**

[18] racheloffline.org 2017. RACHEL Offline. **https://racheloffline.org/**. (2017).

[19] Ali Raza, Yasir Zaki, Thomas Pötsch, Jay Chen, and Lakshmi Subramanian. 2017. xCache: Rethinking Edge Caching for Developing Regions. In *Proceedings of the Ninth International Conference on Information and Communication Technologies and Development (ICTD '17)*. ACM, New York, NY, USA, Article 5, 11 pages. **https://doi.org/10.1145/3136560.3136577**

[20] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. 2001. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '01)*. ACM, New York, NY, USA, 149–160. **https://doi.org/10.1145/383059.383071**

[21] Samuel Sudar, Matt Welsh, and Richard Anderson. 2017. Demo: Siskin: Leveraging the Browser to Share Web Content in Disconnected Environments. In *Proceedings of the 12th Workshop on Challenged Networks*. ACM, 21–23.

[22] Alec Wolman, M. Voelker, Nitin Sharma, Neal Cardwell, Anna Karlin, and Henry M. Levy. 1999. On the Scale and Performance of Cooperative Web Proxy Caching. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*. ACM, New York, NY, USA, 16–31. **https://doi.org/10.1145/319151.319153**