Bringing P2P to the Web: Security and Privacy in the Firecoral Network

Jeff Terrace, Harold Laidlaw, Hao Eric Liu, Sean Stern, and Michael J. Freedman
Princeton University

Abstract

Peer-to-peer systems have been a disruptive technology for enabling large-scale Internet content distribution. Yet web browsers, today's dominant application platform, seem inherently based on the client/server communication model.

This paper presents the design of Firecoral, a browser-based extension platform that enables the peer-to-peer exchange of web content in a secure, flexible manner. Firecoral provides a highly-configurable interface through which users can enforce privacy preferences by carefully specifying *which* content they will share, and a security model that guarantees content integrity even in the face of untrusted peers. The Firecoral protocol is backwards compatible with today's web standards, integrates easily with existing web servers, and is designed not to interfere with a typical browsing experience and publishing ecosystem.

1 Introduction

Peer-to-peer content distribution has been inarguably successful for large file distribution (*e.g.*, BitTorrent [2]). These protocols account for a large fraction of Internet traffic, and they have spawned their own body of research, whether focusing on performance, fairness and incentives, network friendliness, or resource efficiency. Peer-to-peer (P2P) protocols are attractive from the perspective of both content publishers (who can significantly save on upload bandwidth and server provisioning) and potentially even ISPs (who can reduce inter-AS transfers if protocols are ISP friendly [7]).

As of yet, however, P2P services have been restricted to stand-alone applications, not transparently incorporated into Web browsing and seamlessly running over HTTP. If indeed the now-trite statement is true—that "web browsers are the new operating systems", especially considering technologies [3, 13] which further enhance browser capability—then P2P exchanges are once again absent. For the browser is built around the identifying primitive of a domain name. Conflating naming, location, and authorization, browsers use domains both to specify *where* to retrieve content (web objects) and *what security policies* to enforce on downloaded objects. This design conflation seems antithetical to P2P systems.

There have been some attempts [11, 6, 10] to build P2P-like web content distribution networks (CDNs), most notably our own CoralCDN [5]. CoralCDN has been deployed continuously on the PlanetLab research platform for almost five years, receiving requests from around a million users (unique client IPs) per day. However, while CoralCDN was designed to be highly scalable through a DHT-like content discovery mechanism, it always remained on PlanetLab and thus never operated in a more "peer-to-peer" fashion. Not surprisingly, this restricted deployment became a major limitation: CoralCDN's bandwidth demands surpassed PlanetLab's available capacity within its first year, leading us to deploy fair-sharing algorithms that reject large numbers of requests for popular domains [4]. But there's good reason for this deployment restriction, as malicious proxies otherwise could have returned spam, malware, advertisements, or other modified content to unsuspecting web clients.

We are building a new system, Firecoral, that seeks to overcome these limitations and realize P2P web content distribution. With client software implemented as a browser extension, Firecoral allows users to perform multi-source downloading (via HTTP Range Requests) between participating users and thus, in some sense, "share" their browser caches. This paper focuses on several key aspects of Firecoral's design that we believe are important for the wide adoption of a P2P-based web CDN.

For content providers that adopt Firecoral to reduce bandwidth consumption, the system should

- Integrate easily into existing web servers and provide a familiar user experience,
- Support both Firecoral and non-Firecoral clients simultaneously for backwards-compatibility, and
- Not interfere with existing advertisements, analytics, etc. that are in use by servers.

For clients that seek to adopt Firecoral, the system should

- Be easy to install and transparent to use,
- Provide content integrity (security) of web content, even if content is fetched from untrusted peers and

even if its origin server has not adopted Firecoral itself, and

- Respect privacy/sharing policies through powerful, easy-to-use configuration management.

The remainder of this paper is organized as follows. Section §2 presents Firecoral's architecture and protocols, including the cryptographic mechanisms by which it ensures the integrity of P2P content. Section §3 describes how Firecoral allows users to flexibly specify what content they wish to share—via programmable "opt-in" configuration expressions—to manage its privacy implications. We conclude with some performance benchmarks in §4.

Focusing on security, privacy, and usability, this paper is *not* about the two main properties that are of typical interest in the P2P-CDN literature: performance and incentives. While using HTTP Range Requests to specify peer exchanges [9], Firecoral is free to implement any combination of peer-selection and chunk-scheduling algorithms to improve performance. Considering incentives, however, leads to new research questions. Traditional bilateral-barter approaches (such as BitTorrent's rate-based tit-for-tat [2]) are unlikely to work well in this setting, as web objects are often quite small and have very low latency requirements. Incentivization approaches based more on stored value may be more applicable (*e.g.*, PACE [1]), although we leave this research direction to future work.

2 System Architecture

Firecoral enables peers to cooperatively disseminate content by sharing their browser caches. The high-level Firecoral architecture has four main components: *browser clients* (or *peers*), *origin servers* that publish content, *tracker(s)* that store peering information and content metadata, and cryptographic *signing service(s)* that authenticate content. These latter three services can be run by separate administrative entities or in more complex combinations. Peers can interact with Firecoral-unaware origin servers, but any content downloaded from them must be authenticated by a signing service before it can be shared with other peers. Firecoral is not designed to work with completely unmodified clients (unlike [5]), who are otherwise unable to verify the integrity of content downloaded from untrusted peers.

We foresee two main types of trust relationships governing the Firecoral ecosystem. In the first, a tracker trusts a particular signing service to authenticate content *for any domain*. This can be because both logical entities are run by the same party, or because the signing service plays a special trust role in the network (much like certificate authorities such as Verisign do for the Web). Our initial deployment will be of this type: We will operate both a tracker for any domain (likely based out of a cluster at Princeton) as well as a signing service that it trusts (deployed across PlanetLab to satisfy its resource requirements). This design supports P2P exchange even for unmodified origins.

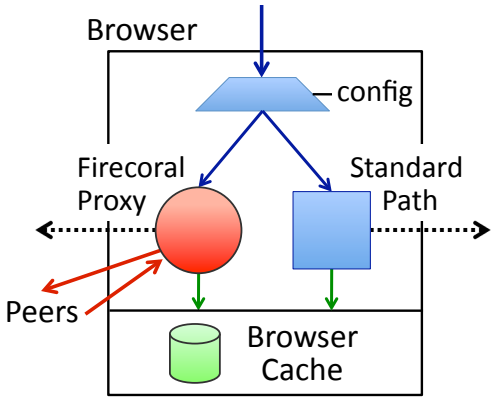


Figure 1: **Firecoral-Enabled Browser Control Path**

In the second scenario, an origin server trusts a particular signing service to authenticate content *for its domain*. This again may be because the origin server itself runs the signing service, or because it again explicitly trusts some third party like Verisign. A third-party tracker may not be aware of such a trust relationship, so it would need to verify the origin's authorization on demand.

In general, we envision multiple different Firecoral deployment scenarios to coexist, much as there exists a vibrant BitTorrent ecosystem: from publisher-run trackers, to access-controlled private trackers, to large centralized trackers such as The Pirate Bay [12] (which serves more than 18M concurrent peers in over 1.5M torrents), and to transient trackers selected from a DHT.[1]

In the next section, we detail the protocol interactions between these four parties.

2.1 Firecoral Browser Extension

We now describe Firecoral's client-side behavior. Firecoral is functionally implemented as a browser extension (currently for Firefox) that operates a local proxy to accept both local and remote HTTP requests. We designed this extension with three goals in mind:

1. Expose the user's web browser cache for remote peer transfers.
2. Provide a powerful configuration interface to allow the user to specify exactly what web content is shared.
3. Guard the user from possible malicious content injected by other users.

[1]This paper does not consider untrusted trackers. Similar to BitTorrent, however, a Firecoral tracker really only needs to be trusted to provide availability, not integrity, at a small additional client-side verification cost.

The Firecoral proxy operates in a background thread within a user's browser. When the user makes an HTTP request, a Firecoral request manager checks the URL against the user's dynamically configured whitelist (further explained in §3) and either forwards the request to the local Firecoral proxy or to the browser's standard processing path. This basic browser architecture is shown in Figure 1. The proxy handles requests coming from the local browser as well as remote installations of Firecoral in peer browsers. The Firecoral proxy uses the same disk cache as the standard browser, preventing Firecoral from wasting additional disk space of the client.

We first consider the simple case that a client accesses a whitelisted domain using a pre-configured tracker. When the Firecoral proxy receives a request from the local browser, it sends the URL to this tracker (step 1 of Figure 2). The tracker responds with one of three possible messages:

Content at Peers. The proxy first receives a list of content chunks that comprise the web object, as well as peers known to be storing the chunks. Next (Step 2 of Fig.2(a)), the proxy attempts to download these chunks from peers in parallel. Upon receiving a download request, the proxies running on these remote peers check both if the requested URL (or Range Request for the URL) is cached locally, as well as if their local sharing policy permits access to this content. If fetching the URL's content from its peers is successful (and content verified against tracker-supplied hash information), the local proxy returns the data to the browser, and registers itself with the tracker as storing the URL chunks (Step 3 of Fig.2(a)).

URL Not Found. If the tracker is unaware of the requested URL, no Firecoral user has yet added the URL to the tracker. So, the proxy attempts to retrieve the content from its origin server (Step 2 of Figs.2(b) and 2(c)) in order to make it subsequently available via Firecoral.

Consider first the case when the origin server operates a colocated signing service. Then, upon receiving a HTTP request from a Firecoral client, it defines a chunking scheme over the URL's web object, signs the object and these chunks, and returns the result in HTTP response headers. (We detail the specific security mechanisms in §2.3.) The client subsequently submits this metadata to the tracker (Step 3 of Fig.2(b)). The tracker in turn verifies the content's signature, retrieving the origin's public key on-demand if necessary (Step 4 of Fig.2(b)) before caching it.

On the other hand, Firecoral is also designed for unmodified origin servers. In this case, the client proxy must still retrieve a valid signature on the content, so after downloading a vanilla web object from the origin server (Step 2 of Fig.2(c)) and returning it to the browser, it forwards a request to a signing service (Step 3 of Fig.2(c)). The signing service cannot just blindly trust that the client submits to it the URL's correct content however, so it in turn retrieves the URL from the origin server itself (Step 4). The signing service applies some chunking scheme over its retrieved content, computes a signature, and returns the results. Finally, the client submits this content to the tracker (Step 5). If the signing service is not explicitly trusted by the tracker, it may query the origin server to verify that the signing service is authorized to speak for it (not shown), information it also caches. Once verified, the tracker adds the URL to its database. Of course, this extended process occurs when a new URL is introduced to the tracker.

URL Expired. Every mapping from URL to content (chunks) at the tracker is associated with an expiration date. If that date has passed, the proxy will try and retrieve a new version of the content from the origin server (or simply a `Not-Modified` message), retrieve a signature, and update the tracker. If the origin server fails, the proxy will attempt, instead, to fetch the content from the peer list of the most recently expired version of the URL.

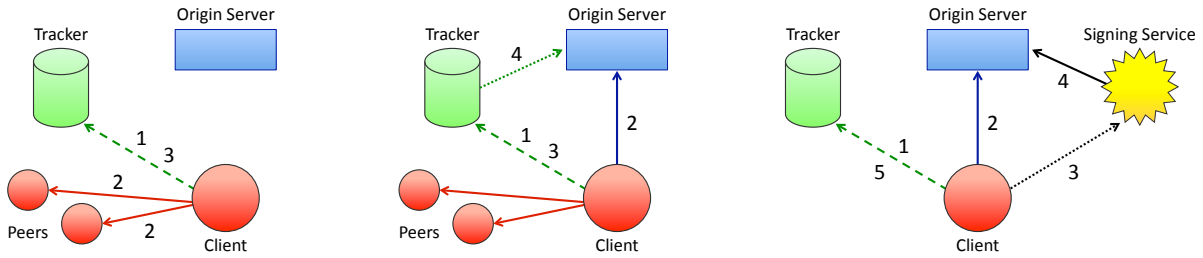
This protocol description was from the perspective of a client-initiated Firecoral request, *e.g.*, the client had the URL whitelisted. (We discuss such configuration further in the next section.) Another strategy for server offloading is driven by server-side adoption: Origin servers can respond to HTTP requests from Firecoral-aware clients with redirection messages to an appropriate tracker.

One implicit security assumption in these protocols is that an adversary cannot hijack DNS or TCP requests; a similar assumption governs non-SSL web security today. Alternatively, one could require that all server-side parties have SSL certificates and that communication between origin/tracker, origin/signer, and tracker/client uses `https`.

2.2 Tracker

The responsibility of the Firecoral tracker is to provide peer discovery to users. The tracker—currently implemented in PHP with a memcached, MemcacheDB, and MySQL backend—maintains a set of known URLs, each mapped to a list of chunks. Each chunk is mapped to the set of peers currently storing the content. The tracker will only accept a new URL mapping if it is newer than the existing one (according to its signature) and if the signature verifies. The signature should be generated by one of a list of trusted signing services (public keys) acceptable for any domain, or by a key approved by the URL domain's origin server (as verified through a `publickeys.txt` HTTP request to the origin, as in Step 4 in Fig.2(b)).

Upon downloading an object chunk, clients report their caching status to the tracker, which adds them to its list of known cache points. When a URL is requested, the tracker returns some subset of each chunks' peers to the client.



(a) P2P Exchanges on Known URLs (b) Retrieving Content from a FC-Aware Origin (c) Authenticating Content with a Signing Service

Figure 2: **Protocol Interactions between Firecoral Participants**

The tracker additionally keeps track of the last time it has received communication from each Firecoral peer. When a pre-configured length of time has passed without any communication (and no persistent TCP connection between client and tracker exists), the tracker removes the client from any chunk it was listed as storing. Unlike typical browsers, however, our client implementation is designed for unreliable peers, so chunk requests are issued in parallel and quickly fail over to handle unresponsiveness.

2.3 Signing Service

The signing service serves as a trusted third-party that validates content hashes. Without this service, malicious users could send arbitrary content to their peers, posing a huge security risk to Firecoral users. As mentioned above, the tracker will only accept URLs that are accompanied with a content hash signed by the signing service. To initiate a request, a client sends $\langle url,\ h_{body} \rangle$ to the signing service, where h_{body} is the SHA-256 hash H of the *body* of the document (fetched earlier from the origin server). The signing service then itself fetches the document, D, located at the given URL, and computes h_D, the SHA-256 hash H of the body of D. To support parallel downloading of larger objects, the signing service also breaks D into a set of smaller chunks. While our current implementation uses fixed-size chunks of 50 KB (except for a final 25–75 KB chunk), it's important to note that the chunking scheme can differ per signing service and per object, as the size of each chunk is fully specified in object metadata. Thus, a signing service could as easily implement semantic chunking, such as through Rabin fingerprints [8].

To allow peers to validate the integrity of individual chunks, the signing service computes the following tuples:

$$h_{chunk,i} = \langle\ start_i,\ end_i,\ H\ (\ D\ [start_i : end_i]\ )\ \rangle$$
$$h_{chunks} = [\ h_{chunk,1}\ ,\ h_{chunk,2}\ ,\ \ldots\ ,\ h_{chunk,N}\ ]$$

The signing service then uses its private signature key, k, to sign the following message:

$$\sigma = Sig_k\ (\ url,\ h_D,\ h_{chunks},\ hdrs,\ t_{now},\ t_{exp}\ )$$

where $hdrs$ is a subset of HTTP headers from the origin server, t_{now} is the timestamp of the request, and t_{exp} is the expiration date of the signature (likely at least as long as the document's HTTP expiry time). Our current implementation uses RSA-PSS signatures with 1024-bit keys. The following is then sent back to the client:

$$\langle\ \sigma,\ url,\ h_D,\ h_{chunks},\ hdrs,\ t_{now},\ t_{exp},\ [D]\ \rangle$$

where D is only included if $h_D \neq h_{body}$. (Note that the prior downloaded content can be displayed to the client's user, but D is replaced in its browser cache, as only this version of the web object has been verified by a signing service and thus valid for sharing.) This design choice, as opposed to having clients always downloading content directly through the signing service, is a trade-off focused on minimizing the signing service's upstream utilization.

The client will subsequently submit this information (minus the actual D) to the tracker, who verifies σ. When other clients subsequently make requests to the tracker, it returns the $\langle start_i, end_i, h_{chunk,i} \rangle$ for each chunk.[2] Then, as a client's Firecoral proxy downloads chunks from other peers, it verifies that $H(chunk_i) = h_{chunk,i}$ before returning it to the client.

3 Configuration Management

So far, we've focused on *how* to incorporate secure peer-to-peer sharing with Firecoral, not *when* to actually do so. Our process balances two sets of conflicting interests.

On one hand, using Firecoral could enable clients to access content that would be otherwise unavailable due

[2] If the tracker is only trusted for availability, but not content integrity, it should also return the actual σ to be verified by the client, but we do not currently implement this case.

to under-provisioned servers, to possibly improve performance given geographic distributions of clients and servers, to circumvent censorship or blocking restrictions, or to gain potential privacy benefits with respect to origin servers by avoiding them directly in many cases.

On the other hand, performance would certainly suffer with respect to accessing content from well-located, provisioned websites or commercial CDNs. Firecoral should not noticeably interfere with existing advertisement or analytics methods desired by servers. Personalized or private data should not be exposed by users, and clients should have flexible and powerful control over what content is retrieved from origin servers and what from Firecoral peers, as the very act of requesting certain content from a Firecoral tracker and other peers has definite privacy implications: they see what URLs you download from them.

```
<div type="story" class="article">
  <div class="body" id="fhbody-3011301">
    <div id="text-3011301">
      The first flying auto... The <a href=
      "http://www.terrafugia.com/aircraft.html">
        Terrafugia Transition</a>, which can
        ...
    </div>
  </div>
</div>
```

Figure 3: **An example of Slashdot's article HTML. The following XPath query would match all external links from articles and comments:** ***//div[@class='body' or @class='commentBody']/descendant::a[starts-with(@href, 'http://')].***

3.1 Enforcement Mechanisms

This section describes the configuration mechanisms by which Firecoral clients can specify their sharing policies. At a high-level, Firecoral allows users to both *whitelist* (opt-in) and *blacklist* (opt-out) of domains and/or URLs.

Whitelisting content on a per-domain or per-URL basis (*e.g.*, `http://*.example.com/*`) means that clients can *a priori* specify a set of domains for which Firecoral should attempt to cooperatively retrieve content from other peers. A client's settings may be based on these domains' chronic under-provisioning, geographic proximity, or content-blocking practices, as well as client privacy preferences. We should note, however, that such a mechanism does nothing to alleviate the phenomenon of *transient* unavailability caused by flash crowds or temporary server failures, which we address next in §3.2.

Blacklisting on a per-domain or per-URL basis allows a client to override the use of Firecoral, either because a server adopts it (per Figure 2(b)) or through dynamic adoption (described next). To avoid server redirection, Firecoral headers are omitted when issuing HTTP requests to blacklisted origins. Firecoral additionally will not cooperatively share (or register with a tracker) content that is marked as `private`, marked as `no-cache`, the result of a POST operation, or some form of similarly personalized content.

3.2 Handling Flash-Crowd Scenarios

The so-called "Slashdot effect" may seem challenging to mitigate transparently. It does not affect the news aggregators or portals for which it is blamed—Slashdot, Digg, BoingBoing, and so on—but rather the often under-provisioned, and almost always unsuspecting, websites to which they link. Whitelisting the portals themselves has no effect, as they have well provisioned and highly redundant server architectures. Requiring adoption by these portals is unattractive from a deployment standpoint. Finally, simply extending whitelisting to `Referer` headers disregards our ad-friendly goals: such a rule captures too many sites (third-party image hosting, CDNs, advertisement networks, analytics services, etc.) in its broad net.

The solution, therefore, seems to be domain- or URL-specific parsing rules that characterize individual links on these pages. One could imagine a simple strategy: A parsing engine applies these rules to content as it is being rendered in the browser. When it discovers a link, it inserts the *full URL* being referenced into the Firecoral whitelist, marking the URL with some short time-to-live value. Subsequent requests to this URL will thus transparently go through Firecoral, rather than directly to its origin server.

Fortunately, modern HTML programming practices offer a nice solution. Many portals or aggregators have a similar textual and presentation styling for their content, such as a vertically-oriented list and synopsis of articles or links. Thus, through classic abstraction, they typically represent these like content elements with a common name—a distinct *CSS class*—that will specify their visual properties. From our perspectives, this enables us to write very simple rules with which to *parse* the HTML pages: Specifically, use an XML Path Language (XPath) query that extracts URLs (in `href`'s) located within a specified Document Object Model (DOM) class, as shown in Figure 3 for an example Slashdot article.[3] These queries are simple to write, highly similar or identical across domains, and only extract the desired links (*e.g.*, those within news or story articles). Thus, sites with a higher likelihood of being transiently under-provisioned can be mirrored and shared automatically by Firecoral users as soon as they appear on a news aggregator site. On the other hand, highly-provisioned sites like CNN—or undesirable domains for privacy reasons—

[3]Coincidentally, this article posting of Jan 12, 2009 caused the bandwidth quota of the linked site to be quickly exceeded, leading to its hosting service to return an error code for the page.

Obj. Size	Latency (ms)	Throughput (resp/s)
500B	6.93	356.76
5KB	7.39	340.99
50KB	9.79	284.65
500KB	13.72	197.72

Figure 4: **Mean latency and throughput microbenchmarks for the Firecoral signing service.**

can be avoided by a client through simple domain blacklisting. Since most users will probably want to maintain similar whitelists and blacklists, Firecoral will support subscriptions for common usage patterns, as is already done in many Firefox extensions (*e.g.*, for AdBlock+ filters).

4 Benchmarks

Our prototype implementation consists of a *tracker* written in 1000 lines of PHP, a *signing service* written in 700 lines of Python, and a Firefox extension written in 7000 lines of JavaScript, XUL, and CSS. Our extension implements a full HTTP proxy server in JavaScript, utilizing Mozilla's XPConnect API to use low-level network functions. Although our deployment is currently limited to Firefox, the Firecoral API is open for implementation in other platforms.

Signing service microbenchmarks. We have implemented a prototype signing service in Python running on Apache's `mod_python`. To test its performance capabilities, we placed varying size files on a web server, installed the signing service on a second web server, and used the Apache Benchmark (`ab`) utility to request the files be signed. All three machines had Intel Xeon quad-core 2GHz processors, connected on a 1GBit network. Figure 4 shows the results of our tests.

Even though throughput is somewhat low, recall that the signing service is only invoked the first time a URL is added to the tracker or when a URL expires. Also, the signing service can be scaled out to multiple servers that share a single private key—increasing throughput linearly with the number of servers. This performance is likely due to Python's overhead; the actual 1024-bit RSA-PSS signature (which uses the native Crypto++ library) takes less than 1.2ms on one core. In the future, we may implement the signing service in a faster environment (*e.g.*, C++).

Browser performance impact. The configuration system of §3 needs to seamlessly integrate into the browser without noticeably impacting performance. The list of XPath configuration rules is indexed by domain name, so when a browser page is loaded, a simple hash-table lookup is performed to test if the domain is in the rule set (or, more precisely, a lookup is performed per subdomain). If a match occurs, the XPath query specified is run on the page's DOM. When evaluating Slashdot's front page, Firecoral processes 50 URL additions into the transient whitelist in less than 3ms; extracting out 15 URLs from digg takes less than 1ms.

5 Conclusions

This paper introduces Firecoral, a P2P web-CDN designed around the idea of turning client browsers into servers and transparently sharing their caches. We specifically focused on issues of security, privacy, and usability, all critical aspects of P2P system design, yet mostly ignored in prior work. An initial implementation of Firecoral appears promising, yet several important problems remain. These range from research questions—such as the choice of peering algorithms and incentive strategies—as well as practical deployment issues such as NAT hole punching and tracker scalability. An alpha version of our Firecoral extension is currently available for download at our website at `http://www.firecoral.net/`.

References

[1] C. Aperjis, M. J. Freedman, and R. Johari. Peer-assisted content distribution with prices. In *Proc. SIGCOMM CoNext*, Dec. 2008.

[2] B. Cohen. Incentives build robustness in BitTorrent. In *Proc. P2P-ECON*, June 2003.

[3] J. R. Douceur, J. Elson, J. Howell, and J. R. Lorch. Leveraging legacy code to deploy desktop applications on the web. In *Proc. OSDI*, Dec. 2008.

[4] M. J. Freedman. *Democratizing Content Distribution*. PhD thesis, New York University, 2007.

[5] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *Proc. NSDI*, Mar. 2004.

[6] S. Iyer, A. Rowstron, and P. Druschel. Squirrel: A decentralized peer-to-peer web cache. In *Proc. PODC*, July 2002.

[7] T. Karagiannis, P. Rodriguez, and K. Papagiannaki. Should Internet service providers fear peer-assisted content distribution? In *Proc. IMC*, Oct. 2005.

[8] A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. In *Proc. SOSP*, Oct. 2001.

[9] K. Park and V. S. Pai. Scale and performance in the CoBlitz large-file distribution service. In *Proc. NSDI*, May 2006.

[10] D. Serenyi and B. Witten. RapidUpdate: Peer-assisted distribution of security content. In *Proc. IPTPS*, Feb. 2008.

[11] T. Stading, P. Maniatis, and M. Baker. Peer-to-peer caching schemes to address flash crowds. In *Proc. IPTPS*, Mar. 2002.

[12] The Pirate Bay. http://thepiratebay.org/, Jan 12 2009.

[13] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *Proc. Symp. on Security and Privacy*, May 2009.