A Secure, Publisher-Centric Web Caching Infrastructure

Paper 185 - 22 Pages

Abstract

The current Web cache infrastructure, though it has a number of performance benefits, does not address many of the publishers' requirements. We argue that web caches should be enhanced to address publishers' needs. For example, caches will need to log client accesses, run scripts to dynamically produce content, and give publishers QoS guarantees. In this paper, we propose Gemini, a publisher-centric web caching infrastructure. Central to our design is the architectural assumption that the global web cache infrastructure will be heterogeneous, like the Internet itself—caches will belong to many different administrative domains and have different functionalities. The heterogeneous aspect of the infrastructure raises several issues. For example, because caches can alter content, traditional end-to-end security mechanisms can no longer ensure the integrity and authenticity of content. In this paper, we study issues associated with designing such a publish-centric caching infrastructure. In particular, we propose a security architecture that protects publishers and caches from each other in a heterogeneous caching environment. In our design, we ensure that Gemini is incrementally deployable and seamlessly interoperates with the existing caching infrastructure. Along with a system design, we also present experience gained from implementation and preliminary performance results.

1 Introduction

Web caching, like other forms of caching that occur at various levels of the memory hierarchy (e.g., hardware, operating system, application), exploits the reference locality principle to improve the cost and performance of data access. This has been especially effective at the Internet level, where large geographic and topological distances separate the producers and consumers of web content. The direct and tangible benefits of web caching include: improved access latency, reduced bandwidth consumption, improved data availability, and reduced server load.

Figure 1 shows a typical web caching hierarchy and its operation. A publisher (P) serves objects (A and B) to clients (C1,C2,C3) via the caching hierarchy (V,W,X,Y,Z). Consider the scenario where a cacheable object A is sequentially requested by clients C1, C2 and C3 respectively, and all the caches are initially empty. The first request by C1 will trigger a transmission of object A from P to C1 via caches V, W and Y. Each of the three caches will store a local copy of A. The next request by C2 will result in a cache miss at Z but a cache hit at W. So object A is transmitted from W to C2 via Z. Finally, the request by C3 will cause a miss at X but a hit at V, and the object is transmitted from V to C3 via X. The result: object A is transmitted

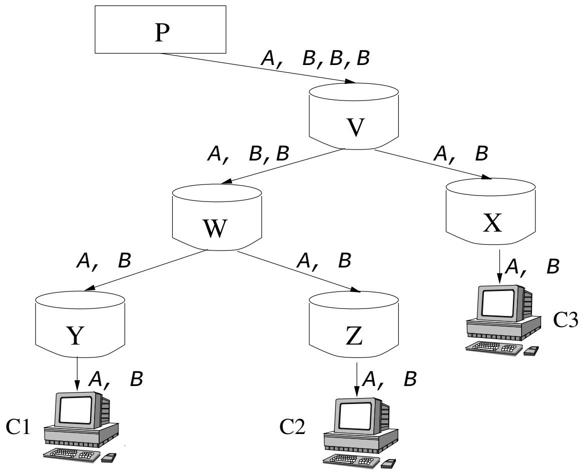

Figure 1: Example caching hierarchy.

across each link exactly once, and C2 and C3 are able to retrieve the object from a cache location closer than the origin server.

The main drawback of today's cache infrastructure is that it is infrastructure-centric, but not publisher-centric. Web caches are deployed and operated by ISPs primarily to reduce network costs and to improve user response times. While these are desirable properties for both ISPs and publishers, from the publisher's point of view, a number of important features are missing in today's cache infrastructure. First, caches are unable to furnish reports on access statistics (e.g., hit counts and click-streams) back to the publishers. This is of particular concern to publishers who rely on accurate hit counts to justify their advertisement-driven revenue model, and to publishers who wish to obtain accurate representations of the size and information consumption behavior of their audience. Second, caches are not equipped to handle dynamically generated content, an increasingly large portion of all web traffic. Today, all customized web content has to originate from the publisher's server, and cannot be reused even by the same client. Finally, caches unilaterally make local copies of web objects, often without the consent or even the awareness of the publishers. As a result, publishers lose control over their objects once the bits leave their web servers. The problem goes far beyond that of copyright infringement. Publishers have no knowledge of the number and locations of cached copies of their objects, making object consistency impossible to maintain. This means that caches may be serving stale or outdated objects to the clients.

For these reasons, many publishers have resorted to cache-busting, i.e., bypassing the caches by tagging their objects 'non-cacheable'. This forces the caches to forward all object requests back to the origin server. For example, object B in Figure 1 is tagged as a non-cacheable object, so each request by clients C1, C2, and C3 is propagated back to P. P has to make three identical transmissions of object B. While this practice assures copyright protection, data consistency, accurate hit counts, and proper dynamic page generation, it also forfeits all the benefits of caching.

We believe that caching is fundamental to the long-term scalability of the web infrastructure, and therefore it is important to realign the interests of publishers and cache operators. We propose Gemini, a publisher-centric web caching infrastructure and paradigm that will encourage the publishers and cache operators to cooperate in the distribution and caching of web content.

The Gemini strategy is to endow cache nodes with communications, storage and processing capabilities that can be beneficially employed by publishers. A Gemini cache node is designed as a next-generation web cache that can transparently substitute for a regular cache, as well as interoperate with existing cooperative caching schemes. A Gemini cache can support a variety of publisher-specified functions. For example, in the data plane, it can support dynamic content generation using versioning (e.g., returning a page based on browser type), filtering (e.g., customized news) and/or other general methods based on sandboxed, virtual machine based languages such as Java. In the control plane, a Gemini cache can support (i) object consistency control, (ii) customizable logging and reporting, (iii) publisher-specific quality-of-service [6] in the form of custom placement and replacement policies (e.g., push, pre-fetch, advanced reservation, and guaranteed object replication), and (iv) access control.

Central to our design is the architectural assumption that the global Web cache infrastructure will be heterogeneous, just as the Internet's routers and links are owned by different administrative domains and different technologies are used to provide connectivity service. In the context of Gemini, we assume that caches belong to many different administrative domains. In addition, they may have different functionalities. For example, Gemini caches that have enhanced functionalities listed above may co-exist with legacy caches in the infrastructure. The heterogeneous aspect of the infrastructure raises several issues. For example, because caches can alter content, traditional end-to-end security mechanisms can no longer ensure the integrity and authenticity of content.

To address these issues, Gemini's design includes three components: a node architecture, a security architecture, and an incremental deployment strategy. Our implementation of the node architecture consists of a modified web cache capable of handling a new document format as well as regular web documents. The new documents, called Gemini documents, can contain Java code written by the publisher, which enables the cache to generate content dynamically. When a request for a document arrives, the cache, which is based on the Squid [8] caching software, no longer simply forwards data to the client. Instead, the cache runs a publisher's code in a Java Virtual Machine (JVM) and logs the request according to publisher directives contained in the requested document. Our security architecture protects the caches and publishers from each other in an environment where caches are distributed across heterogeneous administrative domains. Our deployment strategy allows Gemini to be incrementally introduced to the network. Gemini caches will work seamlessly together with legacy caches and clients.

The rest of the paper is organized as follows. We first describe our security architecture and deployment strategy in Sections 2 and 3, respectively. We then present our design and the prototype implementation of the Gemini node in Section 4. We discuss the performance of our implementation in Section 5. Finally, we will identify related work before we conclude the paper.

2 Security

In traditional distributed communication, end-to-end mechanisms are sufficient to secure communications between the client and the publisher because intermediate nodes do not alter content. In our system, caches are active participants in content generation, so end-to-end security mechanisms are no longer sufficient. But it is not only dynamic content that affects the end-to-end nature of securing content delivery. Caches are now responsible for logging user hits as well. Publishers need assurances that caches will log access correctly, and that these logs will be transported back to the publisher intact. To accomplish this, caches must become fully involved in the system's security.

As an example, consider a publisher's digital signature on a document[1]. Previously, a client would be able to use the signature to verify the authenticity of the document. With Gemini, a cache between the publisher and client might transform the document according to a publisher's instructions, but the cache is unable to alter the publisher's signature because it does not possess the publisher's secret key. The result is that the client is unable to use the publisher's signature to verify the version of the document it receives. The obvious solution to this problem would be to distribute the publisher's secret key to caches, but this has serious ramifications: a cache with the secret key would be able to sign any content whatsoever on behalf of the publisher. Even if the cache's owner is honest enough not to exploit this, crackers who break into the cache may not be as polite.

Our design is guided by four principles:

Protect the publisher as well as the cache. We must provide the publisher with assurance that its content will be handled correctly by caches. Caches need to be protected from malicious publishers.

Publishers decide who to trust. Our system must be as flexible as possible so that the publisher can decide for itself which caches to trust on a document-by-document basis.

Publishers/clients find out about attacks eventually. While it would be nice for any corruption of the publisher's content to be discovered immediately, it is not necessary. This is because the value to a publisher of a single page being served correctly is very small in most situations. Crooked caches do need to be detected within a reasonably short amount of time, however.

The system should be incrementally deployable. As more and more caches, publishers, and clients become Gemini-aware, the system's security should increase, but neither clients nor caches should have to be changed in order to deploy the system.

[1] We consider a "document" to be a single object, rather than a whole "page," or group of objects which a browser might display together.

Along with these four principles, we have created a new trust model. As we have already stated, we allow publishers to choose which caches they trust and to what extent. A publisher who is very concerned either with the correct delivery of content or the correct reporting of logging data might only trust a few carefully selected caches, while a publisher with less to lose might choose to trust every cache. Clients do not trust every cache to produce every document. But if a publisher trusts a cache to produce a specific document, so should a client. Caches trust the publishers very little: they will load and run publishers' content, but only through a sandbox to isolate them from the other processes and content on the cache.

The challenge in securing the cache is to come up with an approach that is both powerful enough to provide protection and generally applicable. For example, one idea would be to require that each cache include a secure coprocessor [18], which is a processor and memory encased in a secure, tamper-proof enclosure. The idea is that all parties can trust the coprocessor to oversee the generation of all content on the cache. Unfortunately, secure coprocessor technology is usually years behind commodity processor technology and more expensive due to the additional engineering and certification necessary to make the device tamper-proof. The resulting lack of performance makes a secure coprocessor unattractive for use in a web cache.

We employ two techniques to enforce the trust relationships outlined above: cache authorization and verification. The first is a way for publishers to explicitly specify which content a cache can generate. One key feature is that a client can determine that the content it receives is generated by an authorized cache. Since the client is the first recipient of the content, it is in the best position to verify that the cache was authorized. And the client machine is often the least-contended-for resource on the path from the publisher.

Our second technique is a way for publishers and clients to verify that authorized caches are performing correctly. This allows the publisher to find out when a cache deemed trustworthy should not be trusted. We cannot prevent a cache from generating content or logging accesses incorrectly. Instead, we use non-repudiation of a cache's output to make establishing which cache is at fault easy. Coupled with random sampling techniques, any cache which misbehaves enough will be caught with high probability. Both publishers and clients can perform sampling to catch crooked caches.

Next, we present the details of our system. We cover the authorization mechanism, the security issues surrounding content generation, and our verification mechanism. Lastly, we consider the other side of the issue, describing how a cache can be protected from publishers.

2.1 Authorization

We rely on a public key infrastructure (PKI) to provide key distribution so that clients, caches, and publishers can check each other's digital signatures. There are several different PKI proposals [4, 15], but they all provide the basic service of associating a public key with an identity. This association is recorded in a certificate, which is a document signed by a certificate authority (CA). Each entity with a certificate can produce more certificates for other entities by acting as a CA.

Each publisher needs a certificate identifying its web site and public key. The format of the certificate is

$$\{P, K_P, Valid, Expires, CA\}_{K_{CA}^{-1}},$$

where P is the publisher's name and URL, K_P is the publisher's public key, from $Valid$ to $Expires$ is the range of time that the certificate is valid, and CA is the name of the certificate authority who created the certificate. The certificate is signed with the certificate authority's private key (K_{CA}^{-1}). Note that we also require each cache to have a public key and a certificate.

A publisher handles cache authorization decisions on an object-by-object basis. Each object includes an access control list (ACL) with which the publisher specifies which caches are allowed to store the object. The format of the ACL is

$$\{URL, K_1, K_2, ..., K_n, Valid, Expires, P\}_{K_P^{-1}},$$

where URL is the name of the object and each K_i is a public key. Each of the keys refer to caches which are allowed to store the object. Instead of a list of public keys, the publisher can also specify a wildcard, indicating that any cache may process the current document. Observe that an ACL is just a special type of certificate, with the publisher acting as the CA.

A publisher can use entries in the ACL in two ways. One way is to authorize a single cache. This is accomplished by including a cache's public key in the ACL. The other way is to delegate the authorization decision to a third party. This is accomplished through a layer of indirection: the publisher includes the public key of the third party in the ACL. Then the third party creates certificates (signed with the key mentioned in the ACL) for caches it wishes to authorize. For example, a company such as Consumer Reports might test caches for functionality and security and might issue certificates (signed by key $K_{approve}$) for those models of caches that meet its criteria. The publisher could mention $K_{approve}$ in its ACL if it trusted Consumer Reports' judgment. A cache that has a certificate stating that it is a model that has been approved by Consumer Reports would then be able to store the publisher's objects. As another example, consider an ISP with many caches. Assume the ISP uses public key, K_{cache} to sign each of its caches' public keys. A publisher could mention all of the ISPs' caches as a group by including K_{cache} in its ACL.

Altogether, a publisher would give the following to a cache: $ACL, \{Headers, Body\}_{K_P^{-1}}$. This is the ACL followed by the object itself. Note that the document and ACL are signed separately since the ACL will need to be passed on to the client. The $Headers$ field contains the URL and directives to the cache about how to handle the object (consistency, QoS parameters, log format, etc.). The $Body$ contains code and data which the cache uses to generate a reply to a client's request for the object.

Along with a response to the client, the cache also includes the ACL. The client is able to check the signature on the ACL and use it to verify whether or not the cache is authorized to produce the requested object. If the cache is not authorized, the client should reject the document. Because the client can retrieve the publisher's certificate, it is able to verify the signature on the ACL and enforce its directives. Unauthorized caches are unable to convince the client that they are authorized.

2.2 Content generation

A cache's reply to the client has the following format:

$$ACL, \{URL, Cache, Client, H(Request), CurrDate, Body\}_{K^{-1}_{Cache}}.$$

Except for the ACL, signed by the publisher, the cache signs the rest of the message: the URL requested, the cache's name, the client's name, a hash of any data sent in the request (e.g. for data conveyed in an HTTP POST message), the current date, and the requested content. There are three purposes for the cache's signature. First, it enables the client to detect tampering with the document on the path from the cache to the client. Second, it tells the client which cache generated the response. This enables the client to be sure that the author of the response is authorized by the publisher's ACL. And third, the cache's signature provides non-repudiation, linking the input (the URL and request data) to the output. The date and the client's name in the message serve to prevent replay attacks, where a third party sends a client stale data. In addition to the above information, the cache needs to provide the client with a chain of certificates which establishes that the cache is authorized by the publisher's ACL. The reason is that the client cannot always determine which certificates are needed to link the cache to one of the keys mentioned in the ACL.

One vital issue is how the cache can send this security information to the client in a manner that does not confuse legacy clients. Note that standard HTTP/1.1 [9] headers already contain the date, the cache name, and the URL. Further, the client computes the hash of the request itself. All the cache needs to send are the ACL, the signature, and the certificates. We include these three items in the HTTP/1.1 Pragma header field. The HTTP specification states that clients and caches which are unable to parse this information will ignore it.

The client, after receiving the cache's response, needs to verify three things: that the cache is mentioned in the ACL, that the request it sent corresponds to the hash of its request in the reply, and that the cache's signature is valid. If there is a problem in any aspect of the response, the client should discard it.

If the publisher desires, the cache can perform access control on the content using standard mechanisms such as username/password pairs, a cookie given to the client by the publisher, or according to the client's network address or hostname. If the publisher believes it is necessary, the cache can even require clients to access private data via SSL [19] or other encryption layer. Standard SSL would not allow a cache to communicate on a publisher's behalf for security reasons, but with the publisher's signed ACL, the client can be sure that the cache with which it is communicating is authorized by the publisher. Unlike our other mechanisms, this one requires that the client to be modified in order for an SSL session with a cache to work.

2.3 Verification

Because a cache signs all of its responses to client requests, it is not able to later deny creating those responses. Any entity with access to the cache's certificate can verify the signature on a response. If a cache were to produce bogus content, its signature would be tantamount to a confession that it was the culprit. A client only needs to present the faulty output to the publisher to prove that the cache misbehaved. Once a publisher is convinced, it can remove that cache from all of its ACLs, preventing the cache from mishandling the publisher documents in the future. The same technique works for catching a cache which fails to report log information. If a client presents the publisher with a signed response from a cache, the publisher can know to expect a log entry from the cache for that response. If the cache fails to return the log entry, the publisher knows that the cache is cheating.

The challenge is in determining when to question the cache's responses. We suggest two schemes: publisher-initiated auditing and client-initiated auditing. Both are based on random sampling so that the more a cache misbehaves, the higher the probability that it will be caught. In client-initiated auditing, the client sets a probability of verifying a cache's response with the publisher. After each response, the client flips a coin to determine whether to query the publisher.

Publisher-initiated auditing involves the publisher using a number of "fake" clients around the network to issue requests for the publisher's documents and return the responses to the publisher. Caches must not know which clients are fake so that the caches do not change their behavior when dealing with fake clients. Note that performance monitoring services such as Keynote[2] already have such clients set up. The publisher can then look at the responses to see that the cache has produced correct output. In addition, the publisher can verify that the fake client accesses were not over-reported or under-reported by caches, helping to assure the publisher that caches are performing logging correctly.

Determining how much auditing should be done is a matter of trading network and cache resources for catching a misbehaving cache more quickly. As the sampling frequency is raised, caches are caught sooner but more system capacity is lost to the sampling process. Finding the right point on this continuum is beyond the scope of this paper.

2.4 Protecting the cache

The security issues discussed so far deal with protecting clients and publishers from malicious caches. However, another concern is publishers who send malicious code to caches. For example, a publisher's code could attempt to access Gemini documents from other publishers or the underlying operating system. Another danger is denial-of-service attacks, that is, code which consumes too many CPU cycles or allocates too much memory. The first problem is similar to the problem of protecting a web browser from malicious

[2] http://www.keynote.com/

Java applets. In either case, a host computer is performing processing on behalf of a publisher. Therefore, we adopt similar protection mechanisms. All of a publisher's code is run inside a sandboxed Java virtual machine so that a cache can have strict control over what operations the code is permitted to perform.

We only expose a few functions to publisher code because we wish to encourage that code to only be used for fairly simple operations such as filtering content according to a user's request. There are no mechanisms for state to be stored from one request to another, and no network communication is allowed. Publisher code is allowed to write logs after each request, but it has no way to read log entries. In total, the API exposed to publisher code consists of functions for performing the following operations: read incoming request headers; write outgoing reply headers; write outgoing data; and generate (a limited amount of) log info for each request. A cache allows the publisher's code to run only when it is satisfying a request. In the future, we may augment these functions with the ability to prefetch data, fetch arbitrary URLs via HTTP, keep state across requests, and even open network connections back to the publisher's server (but not to other network hosts). These additional capabilities will not weaken security significantly, but the risk is that they will encourage publishers to write complicated, resource-intensive applications that will choke the cache.

To counter a denial of service attack, the amount of CPU and memory resources assigned to a publisher's code has to be limited. We rely on operating system-level controls to accomplish this. In addition, techniques like proof-carrying code [13] or software fault injection that examine the code before its actual execution can be applied. Note that since a publisher must sign its documents, the cache can at least identify the guilty publisher if all other security mechanisms fail.

3 Incremental deployment

Having described Gemini's security architecture, we now present our deployment strategy. The Gemini infrastructure is designed to be incrementally deployable and fully interoperable with existing caches, servers and clients. Gemini works with all types of cooperative caching, including hierarchical cache organizations. We have the following design principles:

Cache and document heterogeneity. Gemini caches co-exist and cooperate with legacy caches; not all documents have Gemini versions.

Transparency to clients. Clients need not be modified (except to achieve security).

Transparency to legacy caches. Legacy caches do not need to distinguish between Gemini and regular documents. They can fetch and cache Gemini documents, thereby assisting in their distribution. However, legacy caches will never serve Gemini documents to clients.

Proximity. Gemini content will be served by the authorized Gemini cache closest to the client, which we call the *leaf* cache.

Figure 2 shows the example caching hierarchy with a mixture of regular caches (V,X,Z) and Gemini caches (W,Y). For interoperability the publisher P will have two versions of its object, the regular version B and the Gemini version B', which are identified by their URLs, which we denote URL(B) and URL(B'), respectively. To satisfy client transparency, only URL(B) is publicized. Clients will request and receive B, not B'. In general, clients are never exposed to Gemini documents. Gemini caches are alerted by publisher P of the availability of B' (as detailed in Section 3.2). Specifically, P will provide Gemini caches with the mapping from URL(B) to URL(B'), so that Gemini caches can request and receive B'. Finally, legacy caches can remain completely oblivious to the Gemini scheme and treat B and B' in identical fashion.

To clients and legacy caches, all documents in the system are treated as regular documents. Only Gemini caches (and publishers) understand the associations between Gemini and regular documents. One question remains: how a Gemini cache is able to convert the URL of a regular document into the URL of a Gemini document. We will answer this question next.

Let us illustrate Gemini caching by considering object requests by clients C1, C2 and C3 respectively. We assume all caches are initially empty, but Gemini caches W and Y have been alerted to the availability of Gemini object B'. In response to a request for object B by C1, the Gemini cache Y will perform a mapping from URL(B) to URL(B') and issue a request for the Gemini version B'. Caches W and V will forward the request back to P, and P returns B'. Caches V, W, and Y all make local copies of B'. Note that the legacy cache V does not know or care that B' is a Gemini object; it simply treats it as an opaque object. Now when B' arrives at cache Y, it is used to dynamically generate the object B for client C1. Next, client C2 issues a request for B. A cache miss occurs at the legacy cache Z, but a cache hit occurs at the Gemini cache W. Since W received a request for B (not B') it knows it is the Gemini cache closest to the client. Therefore it executes B' to dynamically generate B and send it to C2. In this case, W may choose to make this copy of B non-cacheable by Z. Finally, C3 makes a request for B and it is propagated all the way back to the publisher. P will send the regular version B and may choose to mark the copy non-cacheable.

In the rest of this section, we describe the methods employed to achieve the design principles above. First, we describe how Gemini interacts with legacy caches in a reverse-compatible manner. Second, we consider how Gemini caches discover and retrieve Gemini documents. As we have said, we leverage the existing legacy caching infrastructure to help deliver Gemini documents. Third, we explain how a Gemini cache determines that it is the leaf for a client. Finally, we argue that this architecture is inherently scalable.

3.1 Discovering Gemini documents

It remains to be seen how a Gemini cache becomes aware of which regular documents have associated Gemini documents and how to find the URL for a Gemini document given the URL of the regular document. The mechanism for this must be robust, lightweight, and of course, it should interoperate with legacy caches. One simple solution would be to define URL naming conventions so that a regular document's name would indicate whether or not it had an associated Gemini document. However, this approach is not robust. If a

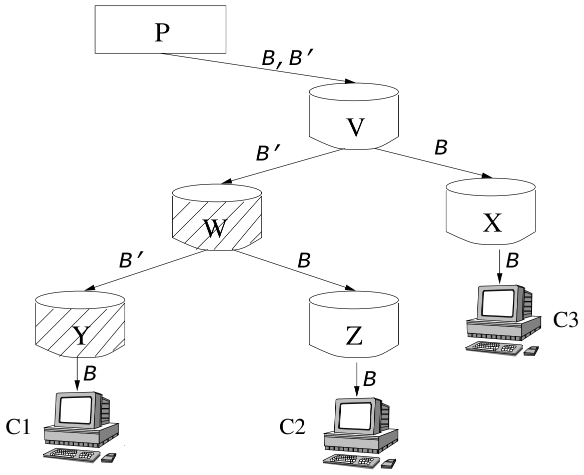

Figure 2: Example caching hierarchy with Gemini caches (shown shaded). The regular version of the document is called B and the Gemini version is called B'.

document's URL inadvertently contained the notation indicating it had an associated Gemini document, a Gemini cache might attempt to load the non-existent Gemini version, leading to increased delay for clients and additional useless requests for the server.

Our approach is to require the publisher to explicitly notify Gemini caches about which regular documents have associated Gemini documents. The notifications are piggy-backed on top of regular HTTP/1.1 responses so that all caches on the path from the publisher's server to the client are notified. Caches store publisher notifications as soft state, throwing away notifications when available space is low or when a notification has not been refreshed recently. Below, we will discuss the format of the notifications, how caches use and manage storage for the notifications, and how they are piggy-backed.

Each notification contains three pieces of information: a server name, a pattern to match, and a transformation to convert a regular document's URL to a Gemini document's URL. A Gemini cache uses the notification as follows: When a request arrives, the cache looks at the URL. The cache finds all notifications with a server name the same as the server named by the URL. For each of these notifications, the cache tries to match the path[3] in the URL against the pattern in the notification. A pattern match indicates that there is an associated Gemini document. On a match, the Gemini document's name is formed by applying the transformation contained in the notification.

In our implementation, the pattern is specified as a suffix match. This is simple and extremely efficient to implement. Also for efficiency's sake, the transformation is implemented as a string to be appended to the regular document's URL. Table 1 is an example of how the cache keeps notifications in a lookup table. The first line of the table indicates that any URL from the server www.a.com which ends with ".html" has a Gemini version. For example, the URL http://www.a.com/index.html would have an associated Gemini

[3]The path is the piece of the URL after the server name and port number.

Publisher	Suffix match	String to append
www.a.com	.html	.gemini
www.a.com	/	index.gemini
b.com	/main/index.html	-enhanced

Table 1: A Gemini cache's lookup table.

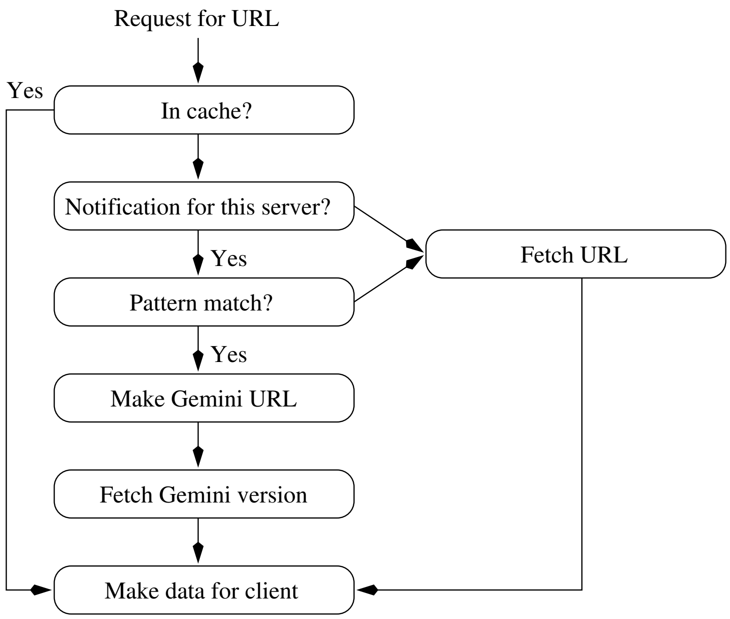

Figure 3: How cache satisfies a request

version named http://www.a.com/index.html.gemini. Notice that it is possible for a single server to have multiple notifications in the lookup table. If a URL should match more than one line in the table, the cache will randomly choose which of the transformations to apply. A cache may wish to limit the maximum number of notifications a publisher can have in the table simultaneously to prevent a resource consumption attack by a malicious publisher. Because entries in the lookup table are soft state, the cache is free to discard them.

Servers include notifications when they reply to requests for regular documents. This is done by inserting an HTTP "Pragma" header. Legacy caches ignore the header, while Gemini caches are able to parse it. Before a Gemini cache has a notification for a publisher, it will not be able to tell which of the publisher's regular documents have Gemini versions. But with the first response from the publisher, the cache will receive a notification. Subsequently, the cache will be able to discover which documents have a Gemini version.

Figure 3 is a flow chart depicting a Gemini cache's procedure for satisfying a client's request. After receiving a request for a URL, the cache first checks to see if it has a local copy of that URL. If it does not, it then checks to see if it has a notification for the server mentioned in the URL. If there is no notification or if the current URL does not match the pattern in the notification, then the cache assumes that the URL is for a legacy document and begins to fetch it. If the URL does match the pattern, then the cache fetches the Gemini version of the document. Finally, the cache produces data and returns it to the client.

3.2 Leaf discovery

The last question we consider is: how does a Gemini cache know when it is the leaf cache for a given request? The Gemini cache which translates a request for a regular document into a request for a Gemini document is the leaf cache. If the cache transformed a request, then it must use the Gemini document to produce a reply for the client. Between the leaf cache and the client initiating the request, all entities expect a regular document. Between the leaf cache and the server, all entities are dealing with the Gemini version of a document. The leaf cache acts as a translator, converting the Gemini version into the regular version.

The authorization mechanism described in Section 2 complicates matters slightly. If every cache could store every document, the leaf cache would be the first Gemini cache a client's request encounters which has the proper lookup table entry. In the common case, this would be the Gemini cache closest to the client. With security, the leaf cache becomes the first cache that both has the proper lookup table entry and is authorized by the publisher.

When a cache, C, initiates a request for a Gemini document, it must also include its credentials in the request. The credentials consist of certificates identifying the cache. This enables an upstream Gemini cache or the publisher to determine whether or not cache C is authorized to process the Gemini document. The cache which makes the authorization decision does not need to verify any of C's certificates. It only needs to make sure that one of the certificates C sent is mentioned in the publisher's ACL. Verification of C's certificates will be handled by any clients for which C produces content. If C should include phony credentials, it will be able to retrieve the Gemini document, but it will not be able to produce a valid signature which will convince clients that it is authorized.

Alternatively, if the upstream Gemini cache concludes that C is not authorized, then it will return an uncacheable reply informing C that it is not authorized. C will store this reply, again in soft state, under the URL of the regular document which was originally transformed into the Gemini document's URL. The current request for the regular document, as well as future requests, will be forwarded without transforming the request into a request for a Gemini document.

When a Gemini cache produces the regular version of a document from a Gemini version, it will make the regular version uncacheable. There are two reasons for a publisher to use Gemini: either because content needs to be dynamically generated (or is otherwise inherently not handleable by legacy caches) or because the publisher is interested in collecting accurate access logs. In the first case, the regular document is inherently uncacheable since it is customized for the client which requested it. In the second case, the regular document must be made uncacheable so that legacy caches do not hide accesses by serving the document without notifying the publisher.

3.3 Discussion

Two properties of our design make it especially scalable. By coexisting with the current caching infrastructure, we are able to leverage thousands of legacy caches to help deliver Gemini documents. Also, observe that the leaf cache's task of producing a regular document from a Gemini document, which involves public key cryptography and possibly running code from the publisher, is a heavy-weight operation. We push this computational burden as close to the edge of the network as possible. Caches in the middle of the network, where resources are under the most contention, will usually only need to forward documents.

4 Node design and implementation

The Gemini node is designed and implemented as an enhanced web cache that supports publisher-centric functionalities such as dynamic page generation, customizable logging/reporting, and security. The design can also accommodate future extensions to support other publisher-defined functionalities. In this section we will discuss some node level design and implementation issues, and describe its operation. The performance of the implementation will be discussed in Section 5.

4.1 Design issues

Several key design choices were made in our Gemini node implementation, including caching platform, runtime language, partitioning of functionality, and document forwarding. We discuss each issue in turn.

Platform We have built the Gemini node on top of an existing caching platform to demonstrate that our architecture is compatible with the existing caching paradigm. We considered Squid [8] and Apache [10] as two candidate platforms. Squid is a single-process, single-threaded high-performance web cache. It works on an event model similar to many simulators: an event (e.g., data or new connection arrival, disk data arrived, socket ready for writing) triggers a call to the appropriate handler code. Apache, on the other hand, is a multi-process web server which can also perform caching. A main process accepts connections and dispatches them to one of several slave processes which return a response to the request. While Apache is easier to program and more robust due to its multi-process structure, it lacks critical features. It uses HTTP/1.0 instead of HTTP/1.1 to fetch documents and does not have ICP [21], the dominant inter-cache communication protocol. In addition, Squid is much more widely installed for the purpose of caching than Apache. For these reasons, we have decided to use Squid.

Runtime language The active component of a Gemini document, such as code for dynamic content generation, should be written in a platform-independent programming language such as Java or Perl. Special care must be taken to prevent the execution of malicious code. Both Java and Perl allow sandboxing, i.e., running code within a special environment such that its access rights to node resources are limited. Since Java has been especially designed for running code in such restricted environments, and considerable research efforts have been spent on enforcing these restrictions, we decided to use Java as the runtime

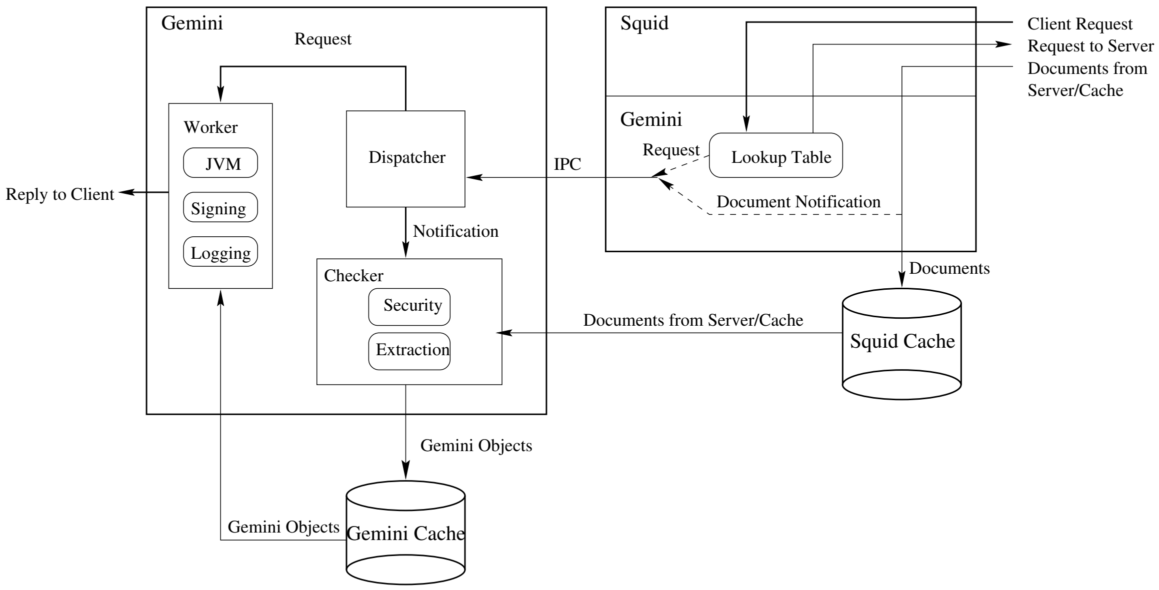

Figure 4: Node Architecture.

language and add a Java Virtual Machine (JVM) to Squid.

Partitioning of functionality We partition the Gemini node design into two processes, as shown in Figure 4. On the right is the Squid process, with three modifications: (i) a lookup table to store soft-state information on the availability of Gemini document versions, (ii) ability to fetch Gemini documents listed in the lookup table but not yet in the cache, and (iii) forwarding of Gemini request (and document if applicable) to the Gemini process. On the left is the Gemini process, which hosts the JVM and security functions. The security functions, implemented in C for performance reasons, are used to verify signatures on incoming documents and to sign outgoing, dynamically generated documents.

We choose to locate the JVM and security functions in a process separate from the Squid process for several reasons. First, Squid's event-driven model makes resource control and accounting difficult. If we invoke a Java routine from within Squid there is no way to limit its running time. This leaves us vulnerable to denial-of-service attacks. Second, this event-driven model increases programming complexity and limits concurrency. In general, high-latency operations are best partitioned into processes separate from the Squid process.

With two processes working in parallel, we can elect one process to be the front-end that accepts client object requests. We choose the Squid process to serve as the front-end for the following reasons: (i) leverage Squid's existing connection handlers; (ii) make the common case fast (there will be a small number of Gemini objects to begin with); (iii) Squid can transparently download Gemini objects using the standard caching hierarchy. For performance reasons, the Gemini process sends a dynamically generated page directly to the client (without going back through the Squid process). This requires the Gemini process to generate the necessary HTTP headers. Interprocess communication (IPC) between the Squid and the Gemini processes is based on a single Unix domain socket.

The Gemini process does not access Squid's cache and index. Rather, it stores its documents in a separate cache and maintains a separate index. This allows the Gemini cache to implement its own replacement policies and support publisher-specific and/or document-specific QoS requirements.

Store and forward document delivery To prevent unnecessary delays, efficient implementations of Web caches store data arriving from a server in their cache and concurrently stream the data to the client requesting it. Our implementation also uses this technique, except at a leaf cache. The leaf has to check the signature of a document before it can send the document to the client. Checking the signature requires the availability of the complete document in the cache.

4.2 Node operation

Now we describe the operation of the Gemini cache node, and explain the interactions between the various node components shown in Figure 4. The Squid front end receives a document request, and in the event of a cache hit, satisfies the request immediately using the cache's copy to produce a reply for the client. In the event of a miss, it performs a table lookup (per Section 3.1) to query the existence of a Gemini version. If the Gemini version exists, and is cached locally, the Squid process will pass the request over to the Gemini dispatcher thread. Otherwise, the Squid process will initiate a fetch of the object using the standard caching hierarchy. When the object arrives, the Squid process caches the document in its cache and hands a pointer together with the original request to the Gemini process via IPC.

The Gemini process consists of three types of threads: a single dispatcher thread, a pool of checker threads, and a pool of worker threads. The dispatcher thread receives requests and documents from Squid and puts them into a request queue and a document queue, respectively, for subsequent processing. The checker threads are assigned to documents from the document queue, and they perform extraction of document parts and signature verifications. Depending on the Gemini document type, further processing is performed. Two types of Gemini documents are supported, active and non-active Gemini documents. Non-active Gemini documents are simply regular documents with appended signatures and the presence of some Gemini headers. These headers indicate, for example, the information to be logged when the document is requested. After checking the document's signature and parsing headers, non-active documents are stored by Gemini in its own cache. Active Gemini documents, on the other hand, may include Java classes in addition to headers and signatures. These classes are extracted and stored in a per-document directory.

The worker threads process the requests from the request queue. The JVM is invoked on the requests for active Gemini documents. It loads the Java class belonging to the document and runs it. The output is the document which will be sent to the client after being signed by the worker thread. Java may also create its own logging string. Alternatively, the standard Gemini logging facilities will log the request. If Java fails for a client request (due to programming errors, excessive resource consumption, etc) the Gemini process will revert control of this request back to the Squid process. In this case the Squid process will handle the request as a regular document without a corresponding Gemini version.

5 Performance evaluation

We have implemented the Gemini node as an enhancement to Squid. In this section, we present our preliminary performance results. Since we are concerned only about the performance of the cache node, we use our own Apache server as the publisher. The machine for the Gemini node is a 550 MHz Pentium III with 128 MB of RAM and 9 GB of disk running Linux (kernel version 2.2). For our JVM, we use IBM's Java Development Kit 1.1.8 with native threads. The server, the cache, and the client are placed on separate machines attached via 100BaseT Ethernet to the same LAN. We conduct three experiments to gain an understanding of how the new functionalities impact the performance. For all of them, we use latency as our metric because we are interested in the potential response time degradation due to Gemini compared to a regular document.

5.1 Lookup overhead

As explained in Section 3, Gemini needs to search for an entry in its lookup table for each request received. Our first experiment is to determine the cost of this operation for regular documents without associated Gemini versions. We examine two cases: In the first case, there are no entries in the lookup table for the server named in the request. In the second case, there are lookup table entries for the server, but the request does not match the pattern specified by the entries. For example, a request might be for a URL ending in ".gif" but the entry's pattern only matches URLs ending with ".html". In both cases, it takes about 50μs for Gemini to perform the lookup operation. Compared with the normal 1 ms to tens of milliseconds required to process a document in an unmodified version of Squid, the penalty imposed by the lookup table is fairly small.

5.2 Active Gemini document

Our second experiment shows how long a request spends in each step during the processing of an active Gemini document. We have instrumented Gemini to timestamp the various processing steps, and we created 10 documents r1,...,r10 with identical content. To perform a measurement, we issue requests for all 10 documents, one after other. We repeat this procedure 10 times for 100 total requests. The very first request serves as warmup and is excluded from the results.

Table 2 lists the steps we are most interested in. We show both the mean and the standard deviation for each of them. They correspond to tasks of the major components of the Gemini process in Figure 4. The "Total" line corresponds to the time elapsed between the arrival of the request in the Squid process and sending the last byte of the reply by the Gemini process. It does not include logging, since logging is performed after the reply has been sent.

We issue requests for two active Gemini documents, one containing simple code and the other containing complex code, in order to illustrate how code complexity affects node performance. The simple code (131

	Ad banner rotation				MyYahoo			
	In cache		Not in cache		In cache		Not in cache	
	Mean	(Std dev)	Mean	(Std dev)	Mean	(Std dev)	Mean	(Std dev)
Extraction	0	(0)	7793	(17.0)	0	(0)	66451	(106.9)
Security	0	(0)	34558	(120.4)	0	(0)	33990	(65.7)
IPC	137	(9.5)	506	(11.6)	146	(4.9)	511	(16.4)
JVM	26740	(1440.9)	26809	(405.3)	261151	(103771.9)	710248	(180804.4)
Signing	9883	(190.7)	9876	(174.4)	9864	(190.1)	9911	(115.0)
Total	38116	(1536.8)	98326	(1104.6)	273149	(103761.1)	878917	(183429.6)
Logging	129	(5.8)	130	(5.3)	135	(7.7)	130	(5.3)

Table 2: Micro-benchmarks for processing an active Gemini document (μs).

lines of Java) inserts random advertising banners into a template HTML page. The complex code (559 lines of Java) generates a per-user customized, myYahoo-like page. Both pages draw their data from the cache rather than from remote servers so that we only measure the overhead from the cache itself.

We present the results both for requests for documents already in the cache and for requests for documents that have to be fetched from the server first. When the documents are in the cache, the first two operations listed in Table 2, extraction and security, are not necessary. As Table 2 shows, security and running Java code are the performance bottlenecks. Signing the reply and document extraction are the next most expensive. IPC and logging, however, are quick.

The cost for IPC in the case of a document that is already in cache differs from the cost for a newly fetched document since. This is due to additional IPC overhead for sending a document notification from Squid to Gemini when a new document arrives in the cache. For the ad banner rotation code, the cost of invoking the JVM is identical for both scenarios. However, for the myYahoo code, the numbers vary by a factor of about three and the standard deviation is also quite high. When taking a closer look at the underlying numbers, we notice that most of them fall into two bins. Twelve of the first 14 requests result in JVM running times of about 780 ms and the next 86 result in running times of about 250 ms. We are currently investigating this behavior to understand its cause. The cost for signing a document is identical for all four cases since it always consists of the same computation, which is based on a fixed-length hash of the document. Note that computing the hash is done while the document is being generated.

In general, by optimizing the security operations, (e.g., by using cryptography routines implemented in hardware), and by applying more-advanced Java techniques (e.g., compiling Java byte code to native code as soon as it is downloaded), we expect the performance penalties due to security and running Java code to decrease.

	3.8 KBytes		18.2 KBytes	
	Mean	(Std dev)	Mean	(Std dev)
Gemini doc (not in cache)	71023	(1985.5)	114176	(4597.8)
Regular doc (not in cache)	6470	(1373.7)	21491	(2430.0)
Gemini doc (in cache)	11353	(216.7)	11395	(161.9)
Regular doc (in cache)	721	(72.6)	1486	(34.2)

Table 3: Performance comparison of non-active Gemini documents and regular documents (μs).

5.3 Non-active Gemini document

Our last experiment shows the performance loss when processing non-active Gemini documents (Gemini documents without code) compared to regular documents. We evaluate two different document sizes (3.8 KBytes and 18.2 Kbytes). The non-active Gemini documents are identical to the corresponding regular documents. As in the last experiment, we prepared 10 identical versions of each document, and then fetched these sequentially 10 times.

From Table 3, we can see that the response time degradation for non-active Gemini documents is about 5 to 15 times compared to the response time for the regular version, depending on the document size. If we examine the times more closely, we find that for each request for a Gemini document, about 10 ms is spent in signing the reply, approximately the same as is shown in Table 2. So, for documents already in the cache, signing the reply is the most expensive, accounting for 90% of the processing time. However, if the documents need to be fetched from the server first, then in addition to signing the reply, document extraction and security check also contribute to the slow-down, as is mentioned before. While it takes a constant time to perform the security check, about 34 ms, the time spent on document extraction varies from 7.6 ms to 36 ms, depending on the document size. This is a performance loss due to disk I/O. It is inherent in our implementation because we do not perform document extraction on-the-fly as a document arrives.

6 Related work

Related work comes from four areas: building distributed caching systems, web security, active systems (agents, networks, and caches), and research on securing active systems. There has been a large body of literature on Web caching architectures (e.g. hierarchical and cooperative) and performance enhancement techniques (e.g. cache routing, push-cache). Gemini caches can work seamlessly in these caching architectures and most of the techniques are equally applicable to a Gemini-enhanced caching infrastructure. Within the last year, several private infrastructures such as Akamai, Adero, and Sandpiper [2, 1, 17] have been built to provide publisher-centric caching services. At the architectural level, the key difference between these systems and Gemini is that they assume all caches are under the same administrative domain, while Gemini assumes an environment where there are heterogeneous administrative domains and heterogeneous nodes (Gemini and non-Gemini). Because of this, Gemini has a strong emphasis on security and incremental deployment issues, which are not addressed in other systems. In addition, Gemini nodes support dynamic content, which, to the best of our knowledge, is not supported in the other systems.

There have been several efforts to bring increased security to the web. These include SSL [19], S-HTTP [14, 16], and the Digital Signature Initiative (DSig) [5]. All three of these protocols provide end-to-end security between the publisher and client, whereas the thrust of our work is in providing security even when a third party is generating content.

Gemini can be viewed as a special type of active network [20] with a focus on content delivery applications. Rather than making a router platform active, we make the cache platform active. In addition, we have a strong emphasis on trust and security issues, and discuss incremental deployment issues in the context of today's caching infrastructure. There are two other related active cache projects. Douglis et al. [7] proposes a highly specialized "macro" language which attempts to separate static and dynamic content in an HTML file. Basically, their scheme allows a cache to store some parts of an HTML file while fetching other parts from the publisher. In contrast, Gemini uses a general purpose language, Java, for data plane operations, and also, it allows publishers to specify control plane behavior. Cao et al. [3] have also enhanced a web cache with a Java runtime in order to allow caches to store dynamically-generated content. They emphasize the cache-centric features of their system: caches can choose which applets to store and how many resources an applet may take up. Further, the security model only considers protecting the cache. Our security model seeks to protect the publisher as well as the cache. Also, we give publishers more control over when and how their content is cached. Finally, we have considered how to deploy our solution in the existing caching infrastructure.

The problem of protecting active content from the host computer on which it is running has been explored, but comprehensive solutions have not been found yet. Moore [12] gives a good summary of work on software techniques and algorithms. The driving application for work in this area is on mobile agents. Work on securing the agent has focused on protecting state acquired at one server from being altered by other servers. In contrast, active content in our system does not alter its state as it moves from one cache to the next. Our main concern is that each cache should execute the code correctly. This is also a concern in the realm of mobile agents, but our problem is a somewhat easier one. The reason is that a publisher in our system is able to know what the output of each cache should be while the owner of an agent cannot know since the purpose of the agent is to gather previously unknown data. Yee [22] has proposed constructing a trusted, secure environment for mobile agents using tamper-proof hardware. Agents executing inside the environment can be sure that they will run without interference from malicious entities. This solution is generally applicable and we have considered using it in our own work. As we have said in Section 2, the disadvantage of using trusted hardware is that its price/performance ratio is extremely unattractive due to engineering and construction costs. Building a high performance web cache using secure coprocessors would be prohibitively expensive.

The problem of protecting infrastructure from mobile code has been well studied. Moore [12] also contains a good survey of work in this area. Approaches come in several flavors: language-level protection and run-time checks [11] and load-time [13] checks. Our work builds on research from this area, applying mechanisms developed for mobile agents, or web browsers, to the domain of protecting caches.

7 Conclusion

We have introduced Gemini, an enhanced caching infrastructure which seeks to be publisher-centric. Publishers are given the ability to dictate how caches treat their content both in the data plane and the control plane. In the data plane, publishers are able to ship code to the caches to generate content dynamically. In the control plane, publishers can specify logging and QoS parameters. Learning from the successful example of the Internet's design, we adopt an architecture that has caches with heterogeneous ownerships and functionalities. To accommodate heterogeneity in an environment where contents can be modified by caches, we present a security model which seeks to protect publishers from caches which mishandle content using two methods: (i) giving the publisher control over which caches are authorized to generate content, and (ii) by providing verification mechanisms. In addition, we describe a deployment mechanism which enables Gemini to seamlessly interoperate with the existing caching infrastructure. We also present a node design which builds upon an existing cache, Squid, to implement Gemini's features.

Our preliminary performance evaluation shows that there are several areas in which Gemini could be optimized, especially in security. Future work includes implementing these optimizations as well as adding more features. For example, we wish to extend the publisher's control so that it can also specify the replacement policy for its content. And introducing additional data types such as streaming media would raise a number of research questions on topics ranging from quality of service to security.

References

[1] Adero, inc. http://www.adero.com.

[2] Akamai technologies, inc. http://www.akamai.com.

[3] P. Cao, J. Zhang, and K. Beach. Active cache: Caching dynamic contents on the web. In *Middleware '98*, 1998.

[4] CCITT, Recommendation X.509, The Directory–Authentication Framework. Consultation Committee, International Telephone and Telegraph, International Telecommunications Union, Geneva, 1989.

[5] Y. Chu, P. DesAutels, B. LaMacchia, and P. Lipp. DSig 1.0 signature labels, using PICS 1.1 labels for digital signatures. *World Wide Web Journal*, 2(3):29–48, 1997.

[6] J. Chuang. *Economies of Scale in Information Dissemination over the I nternet*. PhD thesis, Department of Engineering and Public Policy, Carnegie Mellon University, November 1998.

[7] F. Douglis, A. Haro, and M. Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of USENIX Symposium on Internet Technologies and Systems*, 1997.

[8] D. Wessels et al. Squid web proxy cache. http://www.squid-cache.org.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. IETF RFC 2616, June 1999. Available at http://www.ietf.org/rfc/rfc2616.txt.

[10] The Apache Software Foundation. Apache server. http://www.apache.org/httpd.html.

[11] J. Gosling and H. McGilton. *The Java Lanugage Environment: A White Paper*. Sun Microsystems, Inc., 1996.

[12] J. T. Moore. Mobile code security techniques. Technical Report MS-CIS-98-28, Department of Computer and Information Science, University of Pennsylvania, 1998.

[13] G. C. Necula. Proof-carrying code. In *Proceedings of the 24th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, January 1997.

[14] E. Rescorla and A. Schiffman. The Secure HyperText Transfer Protocol. IETF RFC 2660, August 1999. Available at http://www.ietf.org/rfc/rfc2660.txt.

[15] R. L. Rivest and B. Lampson. SDSI - a simple distributed security infrastructure. Available at http://theory.lcs.mit.edu/~rivest/sdsi11.html.

[16] E. Rscorla and A. Schiffman. Security extensions for HTML. IETF RFC 2659, August 1999. Available at http://www.ietf.org/rfc/rfc2659.txt.

[17] Sandpiper. http://www.sandpiper.com.

[18] S.W. Smith and S.H. Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks (Special Issue on Computer Network Security)*, pages 831–860, April 1999.

[19] SSL 3.0 specification. Available at http://home.netscape.com/eng/ssl3/index.html.

[20] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, pages 5–17, April 1996.

[21] D. Wessels and K. Claffy. Internet cache protocol (ICP), version 2. IETF RFC 2186, September 1997. Available at http://www.ietf.org/rfc/rfc2186.txt.

[22] B. S. Yee. A sanctuary for mobile agents. Technical Report CS97-537, University of California at San Diego, La Jolla, CA, April 1997. An earlier version of this paper appeared at the DARPA Workshop on Foundations for Secure Mobile Code.