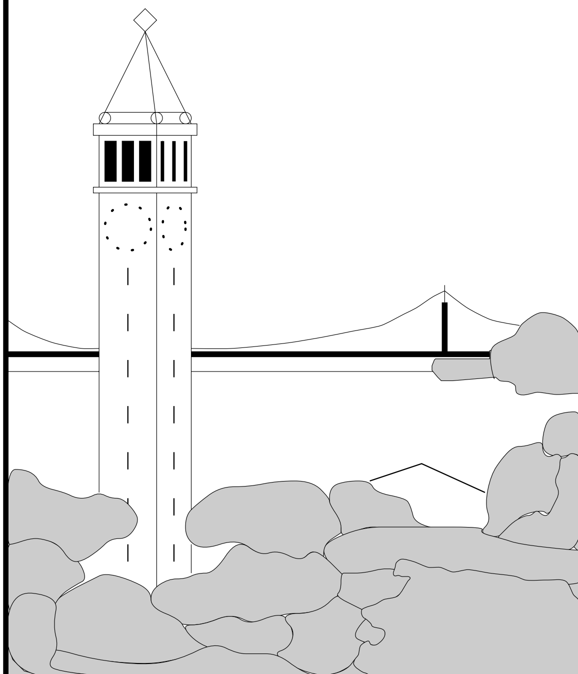
Caching the Web with OceanStore

Patrick R. Eaton
University of California, Berkeley



Report No. UCB/CSD-02-1212

November 2002

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Caching the Web with OceanStore

Patrick R. Eaton
University of California, Berkeley

November 2002

Abstract

We present Riptide, a distributed, cooperative Web caching system. Riptide distributes nodes throughout the infrastructure to serve and manage Web content. Riptide is constructed on the OceanStore distributed storage system and inherits OceanStore's scalability and self-configuration. Nodes may be added or removed as desired. New resources are integrated and begin serving requests automatically; attempts to access unavailable resources are transparently routed to alternative providers. Because the underlying system supports mobile data, copies of content may be moved without expensive directory update protocols. Consequently, the load of serving documents may be quickly redistributed to any of the nodes in the underlying system. We describe the architecture of Riptide and present a preliminary performance evaluation of the implemented system running under a simulated workload.

1 Introduction

Architects of the World Wide Web have turned to a variety of caching techniques to reduce latency and conserve bandwidth [40, 2]. These techniques have proven to mask temporary server and network failures and alleviate network hotspots. Many studies have shown the effects on latency of caching content near clients [22, 4]; other studies have shown the benefits of sharing caches among user populations [11].

Researchers have sought techniques to share content among regional or national user populations with the goal of reducing access latency for all users. Since the presentation of the Harvest web cache [5], a number of the systems have relied on a hierarchy of caches. User requests pass through a proxy which first looks for the content in an institutional-level cache. If the content is found, it is returned immediately to the requesting client; otherwise, the request is forwarded to a regional-level cache. If the regional cache cannot fulfill the request, it forwards the request to the national-level cache. Finally, the national cache responds with the content from its own cache of documents or after retrieving it from the origin server. The result is a global-scale cooperative cache.

Although hierarchies may be effective for caching, they face a number of deployment challenges [31, 38]. First, server management requires significant manual coordination, often across administrative domains. Second, each level of hierarchy introduces delay. Third, caches near the root of the hierarchy may become bottlenecks or add long queueing delays.

In response to these weaknesses, researchers have proposed distributed caching systems [38, 27]. While these proposals solve some of the problems of hierarchical caching techniques, others remain. First, the distributed caches still require significant manual administration. Further, from a performance standpoint, these "distributed" caches still have a notion of hierarchy buried within their design. This ultimately impacts performance: either a hierarchy of directories must be kept updated or caching hints must traverse a hierarchy of participating caches. Nodes high in the hierarchy remain potential bottlenecks.

In this paper, we explore an alternative, namely to exploit recent work in peer-to-peer object storage

systems to perform Web caching. We present Riptide, a distributed Web proxy caching system built on top of OceanStore [23, 30], a global-scale distributed object store. From OceanStore, Riptide gains self-configuration and automatic management. Further, Riptide takes advantage of properties of the underlying system to remove all notions of hierarchy from its design. Riptide is based on the idea of deploying active cache managers throughout the infrastructure. These managers work greedily on behalf of nearby clients to improve the quality of service for the local user population. Local optimizations include increasing levels of replication for popular documents and migrating documents from overloaded servers to balance load. Since OceanStore is a global-scale system, however, the work done by a local manager can still be leveraged by other managers in the system to benefit other populations.

We will describe the design and implementation of the Riptide prototype. We demonstrate that the system can operate in the traditional proxy caching manner or as a push-caching system. We examine the performance of our prototype implementation when confronted with a simulated workload. The resulting system is able to adapt as new services are added to the network and to distribute load across nodes in the network.

1.1 Organization

The rest of the paper is organized as follows. Section 2 provides the context and motivation for our design. Section 3 describes the relevant features of OceanStore, and Section 4 describes the architecture of Riptide. Section 5 describes the state of the current prototype, and Section 6 provides some initial performance measurements. Section 7 presents the related research; Section 8 describes future work; Section 9 concludes.

2 Context and Motivation

Riptide is comprised of many agents distributed throughout the network. These agents cooperate with one another to cache web content close to where it is being used and to keep this content up-to-date. As discussed in Section 4, some of these agents are co-located with client browsers while others reside in the infrastructure.

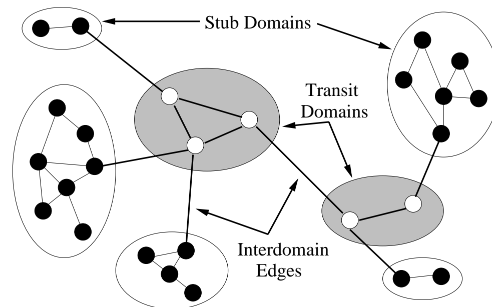


Figure 1: *The Transit-Stub Network Model:* Transit nodes are grouped together to form highly-connected transit domains. Groups of connected stub nodes form stub domains that are connected to transit domains by inter-domain edges.

The main goal of Riptide is to improve the quality of service for clients browsing the Web. An important secondary goal of Riptide is to achieve this improvement with as little management overhead as possible. Riptide achieves these goals simultaneously through adaptive mechanisms that exploit the *bandwidth* and *locality* properties of the physical infrastructure.

2.1 Management of Replicas

Many Web caching techniques involve static placement of proxies and content distribution servers. Unfortunately, static schemes suffer from several deficiencies. First, they are not resilient to failure. Clients may lose service completely when a statically configured resource (such as a proxy cache) fails. Second, static systems may not behave well under load. In order to avoid susceptibility to flash-crowds, static systems must often be over-provisioned. Finally, if the number of replicas is statically over-provisioned to handle high load, then the bandwidth required to keep these excess copies up-to-date would be exorbitant; this tends to discourage use of push-based update of caches.

Riptide utilizes the OceanStore infrastructure to

permit a fluid assignment of server resources to replicas of Web content. Replicas may be transparently created, moved, or destroyed to meet the constraints of the system. Since clients utilize a fault-tolerant, decentralized routing mechanism to locate Web content, they see a graceful degradation in quality of service as resources fail or are reassigned.

2.2 Locality and Transit-Stub Topologies

The global Internet appears as a *transit-stub* topology [48], with clusters of tightly-coupled nodes that are connected via low-dimensional inter-domain links. See Figure 1. Clusters at the edges of the network, often called "stub domains", consist of many individual clients and servers connected via low latency ($<$ 1ms – 10ms), high-bandwidth ($\geq$ 100 MB/s), redundant links. A typical stub might include all of the the systems and networks within a building or organization. Note, the size of a stub domain is determined by connectivity characteristics of the network, not vice versa. A stub domain may be just a few machines in a small home network at the end of a cable modem, a few dozens of computers connected in a LAN, or several thousands of computers spread across a university campus.

While traffic within a stub is virtually unconstrained, traffic between the stub and the rest of the world is often greatly restricted. Latencies between different stubs are often 100ms or more, and the *total* extra-stub bandwidth might be greatly restricted (*e.g.* 100MB/s total for all outside communication). Further, extra-stub bandwidth is typically not free. As a result, organizations may place packet shapers or other traffic bottlenecks along transit links. These bottlenecks further increase perceived latency and decrease available bandwidth.

Another important goal, therefore, is to attempt to place caches of information in the same stubs as the clients that are using this information. Also, since the topology of the network has somewhat more structure than just stubs and inter-stub links, a second-order optimization would seek to place cached information in an adjacent stub when the current stub could not accommodate additional information.

Further, traditional Web caching policies often mark data that is updated frequently as *uncacheable*. Thus, an important additional optimization would be to place replicas of popular, "uncacheable" content within the stubs that it was accessed and *push* updates to these replicas [16, 32]. For popular items, the result would consume much less inter-domain bandwidth than continuous polling.

2.3 Reduction of Server Load

Overcoming limitations in network bandwidth and latency are often not the only justifications for Web caching technologies. In many cases the response time from an overloaded server is many times the round-trip network latency to that server. In fact, extreme server overload occurs during *flash-crowds*, unpredictable traffic surges that swamp servers of popular content. Thus, another important goal for a Web caching architecture would be to transparently adapt and shield clients from the effects of overloaded servers. For rapidly changing content, such a shield may require efficient, push-based updates of data.

3 The OceanStore Infrastructure

In this section, we briefly introduce some of the technologies that enable Riptide's design. Riptide is built on top of OceanStore [23, 30], a global-scale persistent storage system. OceanStore provides a serverless infrastructure that operates without single points of failure to provide continuous access to stored information. To improve performance, data is allowed to be cached anywhere, anytime; OceanStore calls this feature *promiscuous caching*. A data location infrastructure tracks the location of data as it migrates through the system. Continuous monitoring enhances performance through pro-active movement and replication of data. Further, cached replicas of a given object organize themselves into a multicast tree to facilitate timely dissemination of updates.

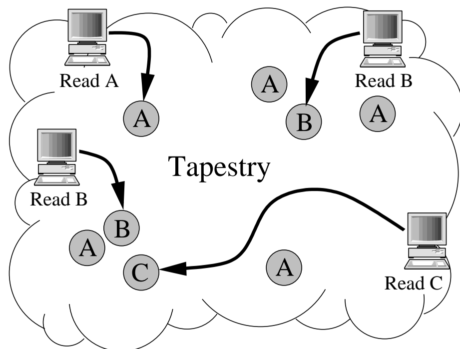


Figure 2: *Decentralized Object Location and Routing (DOLR):* Messages are addressed to *objects* (shown here as discs) rather than IP addresses. DOLR infrastructures such as Tapestry provide *deterministic* routing – guaranteeing to find objects if they exist. They also provide *locality* – utilizing local resources and finding local objects whenever possible.

3.1 Secure Hashes for Naming

Riptide relies on the ability to uniquely identify Web documents and network services. We will refer to these as generically as *objects* in the future. OceanStore employs secure hashing to create *globally unique identifiers* or GUIDs. A GUID is constructed by applying a secure hashing algorithm over the concatenation of the public key of the principal owning the object and the name of the object. In Riptide, we use SHA-1 [25], but the system has been designed to permit the use of any hashing algorithm.

3.2 Distributed Object Location and Routing

Peer-to-peer researchers have begun to explore distributed object location and routing (DOLR) services [17, 28, 35, 37]. DOLR systems are overlay networks that offer a distributed framework in which objects that are named by GUIDs can be located quickly. Since information about these objects is distributed throughout the system, messages are routed to objects by passing them from node to node until they reach their destination.

OceanStore is built on top of the Tapestry DOLR [17, 50]. OceanStore also uses attenuated Bloom filters [29] for probabilistic local-area routing. The combination of Tapestry and attenuated Bloom filters provides locality in routing. If a DOLR is said to exhibit good locality, then messages will be routed over a minimal overlay path to their destination, and when objects are replicated, requests for that object will be routed to the nearest copy of that object. Figure 2 illustrates this idea.

Locality is important for a number of reasons. First, routing locality improves reliability and availability. As the distance traversed by a query increases, it is more likely that the query will be lost or corrupted or will encounter a network partition. Second, finding local replicas of data without routing outside the local area reduces the latency to access documents. Since local-area network latencies are a few milliseconds while wide-area latencies can stretch to hundreds of milliseconds, locality can have a tremendous impact on the latency observed by users. Finally, locality optimizations reduce the bandwidth consumed by a query. By using only local routes to find local objects, the system minimizes the number of messages that must use bandwidth-constrained interdomain links that contribute to the bisection bandwidth of the network. Keeping routes in the local area allows the network to support more simultaneous operations.

3.2.1 Tapestry

The Tapestry DOLR provides object location and message routing services by maintaining a distributed index of all objects and services in the network. The routing algorithms and index maintenance procedures are designed to exploit locality. Tapestry provides algorithms to automatically insert and remove nodes from the infrastructure [17]. Further, it automatically routes around failed nodes and network links.

Tapestry identifies the documents and services in its index by GUIDs. When a node wishes to insert a document or provide a service, it performs a *publish* operation, causing Tapestry to update its distributed index to record the location of the new ob-

ject. To find an object, Tapestry searches the index for the GUID using a variation of prefix-based routing; that is, Tapestry searches the index by incrementally routing toward the object resolving one digit of the GUID at a time. Although worst-case behavior of Tapestry may involve a number of network hops that is logarithmic in the number of participating nodes, Tapestry locates objects much more quickly in practice [50].

3.2.2 Attenuated Bloom Filters

Attenuated Bloom filters [29], a variant of standard Bloom filters [3], are used to provide fast, probabilistic routing in the local area. A probabilistic routing algorithm is one that finds objects quickly when it can and fails equally quickly when it cannot. A hybrid routing framework consisting of attenuated Bloom filters on top of Tapestry finds local objects quickly and remote objects efficiently [29]. In OceanStore, attenuated Bloom filters utilize a self-organizing overlay network that exploits the locality properties of Tapestry; document publishing consists of a compressed wave of information that propagates to a small radius in the overlay network. From the standpoint of this paper, the hybrid combination provides a self-organizing DOLR with good routing locality.

3.3 Replication and Update Dissemination

An efficient DOLR with routing locality provides great flexibility to place replicas anywhere, anytime. For Riptide, this means that any participating node in the DOLR can hold Web content. From a management standpoint, this has a number of important advantages. For example, Riptide may adjust the number of replicas to match the popularity of a document; popular documents could be widely replicated to distribute the load of serving them.

One challenge with replication is keeping replicas up-to-date. OceanStore provides an important mechanism to assist in updating replicas. All replicas for a given document are tied together into a document-specific multicast tree, or *dissemination tree* [23, 6].

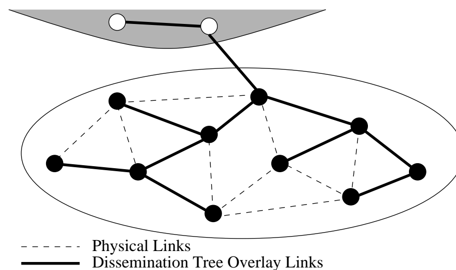


Figure 3: *Dissemination Tree:* OceanStore connects replicas in a multicast tree. The system uses the overlay tree to efficiently disseminate updates to the replicas.

This tree is self-organizing; whenever a new replica is created in the system, OceanStore uses the DOLR to locate one or more additional replicas to use as parents for the new replica. Figure 3 shows how a dissemination tree may be constructed to connect a number of replicas.

The dissemination tree allows the system to forward updates to all replicas of a given document. From the standpoint of Riptide, the dissemination tree provides a natural mechanism for push-based content distribution.

3.4 Comparison to Content Distribution Networks

Content distribution networks attack many of the same problems and share many of the same goals as Riptide. Indeed, companies like Akamai and Digital Island have developed successful businesses by solving these problems. In this section, we will describe briefly how the approach taken by Riptide augments the capabilities of traditional CDNs.

A content distribution network (CDN) is a collection of servers distributed throughout the network. Content from origin web servers is replicated on the non-origin servers available in the CDN. By offloading work from the origin servers and distributing replicas throughout the network, the CDN strives to serve content to clients faster than the origin server could serve the content [21, 19]. Most CDNs operate by using the DNS system to redirect requests from the

origin server to servers in the CDN.

A content distribution network can increase its reach and effectiveness by increasing the number of points in the network that contain servers. As a CDN increases the number of replicas that it manages, the administration burdens increase. Furthermore, there are many places in the network where a CDN simply cannot place a replica. For example, it is unlikely that a CDN would be permitted to place replicas inside of a corporate or university network. We will show that by using new advances in peer-to-peer research, administrators inside these closed domains can add Riptide agents to the network that provide benefits to the local user population while still cooperating with the larger network outside of the organization.

CDNs usually operate their own authoritative DNS name server that manages the address mappings for all machines in their network. These name servers respond to DNS requests with a reply that contains a TTL of a low value. Thus, the CDN operator can change the redirection function in their network relatively quickly [21]. As we will show, Riptide does not rely on the DNS system to locate replicas in the system. In fact, by using a distributed location and routing infrastructure as discussed in Section 3.2, we are able to update the mapping to reflect new resources nearly instantly.

4 The Riptide Architecture

OceanStore provides the abstraction of a large, distributed storage infrastructure. To create a web caching application, we view the storage infrastructure as a large, virtual cache in which to store copies of web content. All nodes that participate in the OceanStore storage infrastructure, not just nodes serving roles specific to the web caching, can store content for retrieval by the web caching application.

In this section, we present the Riptide architecture to manage this large, virtual cache. First, we describe the format of web content stored in OceanStore. We then provide a description of the components that interact with the storage infrastructure. Finally, we discuss how the components cooperate to provide better service to clients browsing the web.

Riptide works by migrating Web content into OceanStore. Clients first search for documents in OceanStore. If the requested content is available through OceanStore, the DOLR routing framework will locate it quickly. Otherwise, clients access the content directly from the origin Web server while simultaneously requesting that the content be placed into the caching infrastructure for future access.

4.1 Storing Web Content in OceanStore

Before web content can be read from the Riptide web cache, it must first be read from the origin server and copied into OceanStore. In the OceanStore infrastructure, the copy of the content is named by a secure hash of the URL of the document. The data that is actually stored in the OceanStore data object is the concatenation of the HTTP header and the document's content. By storing the original response header along with the document's content, agents accessing the Riptide web cache do not need to recreate a valid HTTP header for each document; they can simply read the all of the contents of the OceanStore object.

Riptide does, however, treat one piece of information in the HTTP response specially. When storing an HTTP response into OceanStore, Riptide records the time of the response in the metadata of the OceanStore data object. By storing this piece of information in the metadata, Riptide can take advantage of OceanStore mechanisms to determine if the cached content is fresh enough to serve to the client without reading the content itself. It is important that Riptide be able to evaluate the freshness of cached content in order to maintain compliance with HTTP's cache control requirements.

4.2 Components

Riptide is comprised of many agents distributed throughout the network that interact with virtual, distributed caches. Each agent is of one of three types: browser proxy, gateway, or cache manager. *Browser proxies* are located on the same machine as

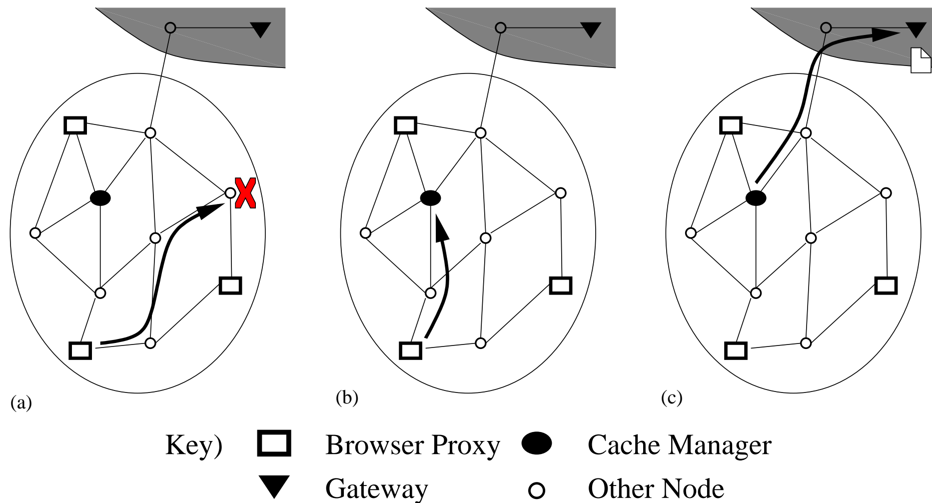


Figure 4: *Inserting a document into the cache.* (a) The browser proxy uses the DOLR to search the cache for content. The search terminates without locating the requested object. (b) While retrieving the document from the origin server, the proxy notifies a nearby cache manager that the document is unavailable. (c) The cache manager sends a request to a nearby gateway to insert the document in the cache.

a user's web browser; proxies intercept requests from the browser and search the cache for a copy of the requested content. *Gateways* are services distributed throughout the infrastructure and are responsible for reading web content from content providers and storing it into OceanStore. *Cache managers* are services distributed throughout the infrastructure and are responsible for managing replicas of Web content to provide better service for clients in their local area. Figure 4 shows how the three types of agents cooperate to insert information into cache; the process will be discussed further in Section 4.4.

4.2.1 Content Caches

Technically a part of the OceanStore infrastructure rather than a component of the Riptide architecture, the *content caches* provide storage for replicas of web content. OceanStore stores objects both in memory and on disk. Every node participating in the OceanStore infrastructure, not just those acting as Riptide agents, serves as a content cache. By default, OceanStore manages storage on individual nodes using the LRU replacement policy. Because the content cache are really nothing more than OceanStore storage nodes, the Riptide content caches also manage storage using the LRU replacement policy. (There are interfaces to allow applications to manage stored objects using different policies, but those interfaces are not used in this work.)

4.2.2 Browser Proxy

The *browser proxy* is an agent that works on behalf of a user's browser to retrieve Web content. The browser proxy can be used in two different configurations. In the first configuration, a proxy is installed on the same physical machine as the client's web browser. In this configuration, the client can communicate directly with the OceanStore infrastructure. In an alternate configuration, the browser proxy can be deployed in the infrastructure. In this configuration, clients forward web browser requests to the browser proxy as in a traditional proxy configuration. The client then relies on the browser proxy on another machine to communicate with the OceanStore infrastructure. Throughout the remainder of this paper, we will assume that the browser proxy is installed on the

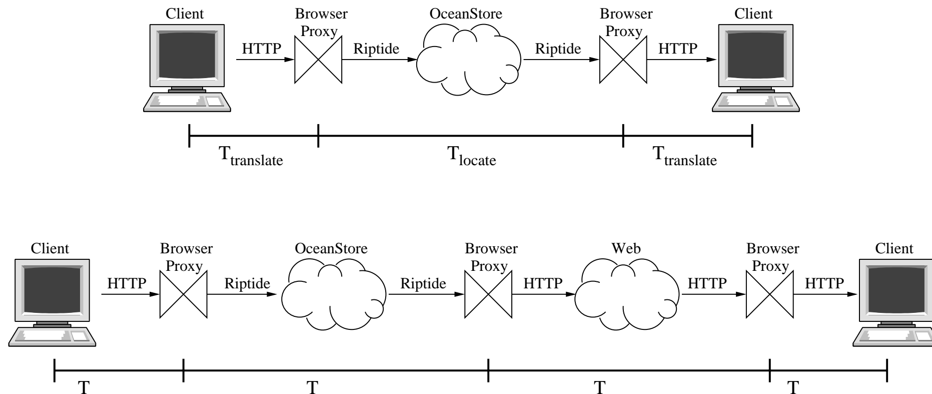


Figure 5: *Timeline of Browser Proxy Operation.* The top timeline shows the operation of the browser proxy when the requested document is cached in Riptide; the bottom timeline shows the operation of the proxy when the document cannot be found in the cache. The small labels above the arrows indicate the message protocol used at each stage of the operation.

client machine.

The operation of the browser proxy, shown pictorially in Figure 5, starts by intercepting an HTTP request from the browser. The browser proxy then makes a preliminary decision about the cacheability of the requested object. For example, the proxy may surmise that requests that contain cookies, reference CGI scripts, or contain variables embedded in the URL are uncacheable. If the proxy deems that the requested document might be cached, it translates the HTTP request into an OceanStore request and dispatches it to the system. If the URL indicates that the document may not be cached, it forwards the original HTTP request to the origin server.

During the translation process, the browser proxy hashes the URL of the requested document to form the GUID for the OceanStore request. If the HTTP request includes any additional restrictions, such as the freshness of the data, the proxy translates these specifications into one or more read predicates. If OceanStore finds the document in the cache, it will return a copy of the content to the proxy; if OceanStore cannot find a copy of the requested content or cannot find a version of the requested content that satisfies the read predicates, it will return a failure response to the proxy. If the system returns a successful read response, the proxy converts the response into an HTTP response and serves the data back to the browser. If the browser proxy receives a failure response, it requests the document directly from the origin server.

The browser proxy uses several timeouts to ensure that requests to the cache fail fast without adding too much latency to the request. This corresponds to keeping T_{fail} shown in Figure 5. First, on all cache read requests, the browser proxy specifies a time-to-live (TTL) of a small number of hops. If after a number of hops equal to the TTL field, the OceanStore DOLR has not found a copy of the content, it will return a failure response to the browser proxy. Because of the locality properties of the DOLR discussed in Section 3.2, the first few hops will usually be in the local area and thus very fast. The second type of timeout used by the browser proxy is a simple timeout alarm. If the request times out before the proxy receives a response, the browser proxy will forward the request on to the origin server.

The browser proxy also communicates with other agents to improve the performance of the system. This interaction will be described in Section 4.4.

4.2.3 Gateway

Gateways are services distributed throughout the network. Each gateway advertises its service by publishing a well-known GUID that identifies the node as a gateway service provider. Others can locate a gateway, then, by routing a message through the OceanStore DOLR to the well-known identifier.

The main role of a gateway is to add new content to the infrastructure and update cached content that is stale. Gateways read content from the origin servers of a content provider using standard HTTP. Examining the header of the response, a gateway determines whether the document may be safely cached. If the document may be cached, the gateway updates stale copies in the cache or inserts a copy of the content; otherwise, the gateway discards the content.

As will be discussed in Section 4.4, gateways use hints from users to determine what content to examine and consider for caching. More proactively, gateways can crawl the web using techniques similar to those used by search engines to search for new content to cache. They could also employ models to predict which documents are likely to be accessed from previously cached pages [22, 26, 13]. The current Riptide prototype uses only hints from users to determine what content to cache; exploring the proactive techniques is an area of future research.

Any ISP or corporation wishing to serve as a web cache provider could deploy gateways in the network. In fact, it might be advantageous for smaller companies or organizations to deploy gateways to ensure that content of local import is available in the cache. Each cache provider would use a different well-known GUID to identify their gateways and would use a different public key to create the names of their documents. While the browser proxy would contain keys for a number of default cache providers, by defining a set of cache provider keys to use, a user could configure their browser proxy to search only the caches owned by providers that they trust or know are nearby.

While any OceanStore application, including the browser proxy, can add objects to the cache, there are several reasons why our design delegates that task to a specialized component. Most importantly, the architecture calls for a specialized component to match the trust assumptions of the underlying OceanStore system. OceanStore assumes a fundamentally untrusted infrastructure. Because we assume untrusted clients will be operating in the infrastructure, we cannot trust that content stored in the cache by clients has not been altered or manipulated. Instead, we create another type of agent that is to be controlled by larger and more respected organizations to insert content in to the cache. By marking content with the key of the trusted organization, clients have greater assurance to the integrity of objects stored in the cache. Another reason is that creating and updating objects in OceanStore requires several digital signatures. While the computation required to create those signatures is not excessive, we wanted to remove the complexity and computational load from the browser proxies, especially those running on thin clients. Finally, recall from Section 3.1 that OceanStore uses the public key of the object's owner to create the name, or GUID, of an object. Thus, to find content in the cache, a browser proxy must know the public key of the principal that inserted the object. The opposing goals of maintaining the integrity of the key pair and making the public key well-known are most tractable when the private key is known by only gateways controlled by a single cache provider.

4.2.4 Cache Manager

While the gateways and browser proxies provide the vital functionality of the system, namely the ability to write content into the cache and read content from the cache, the system does not yet contain any components to adapt the service to the needs of users. The cache manager fills this void by directing the number and location of replicas to improve the level of service seen by local clients. We will describe specifically how the cache manager responds to user access patterns in Section 4.4.

Like gateways, cache managers are services distributed throughout the network. Each manager advertises its service by publishing a well-known GUID that identifies the node as a cache manager. Others can locate a cache manager, then, by routing a message through the DOLR to the well-known identifier.

Because cache managers are most useful to those nearest to them, we envision that the cache managers will be administered by companies or organizations wishing to improve the service their clients receive. As an organization grows or moves people about, they can move the cache managers to serve better their members.

Note, though the cache managers may cache some content directly, this is not their primary responsibility. The cache managers are responsible for directing the creation and migration of replicas near the manager to improve service to local clients. Content is actually stored throughout the underlying system. In fact, because OceanStore supports promiscuous caching which permits any object to be cached anywhere at anytime, web content can be cached even on nodes that are not explicitly providing services to the web caching infrastructure.

4.3 Management Mechanisms

Riptide leverages the fault-tolerance and location-independence properties of OceanStore. In this section, we describe how Riptide uses these features to ease the burden of maintaining caches and configuring clients.

Because Tapestry supports automatic node insertion and removal and efficient service discovery, administrators can easily deploy the Riptide cache and modify a deployed system without configuration changes. To expand the reach of the web cache to new client populations, administrators simply insert a new cache manager into the network. Tapestry will discover the new cache manager and begin to route client requests to the new agent. Clients in the area will initially continue to retrieve content managed by more distant cache managers. As the new cache manager is able to create more replicas in the local area, Tapestry will route client requests to the new, closer replicas. The service can grow evolutionarily and no client configuration changes are ever necessary.

Furthermore, organizations can receive the benefits of cross-organizational cache sharing without the challenges of managing a hierarchy across administrative domains [47]. Because Tapestry can route to content stored anywhere in the network, cache managers can create replicas from copies of content stored in other domains.

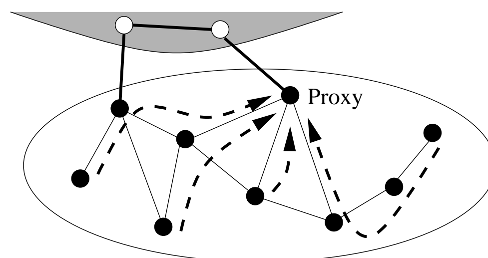


Figure 6: *Shared Proxy Architecture:* All clients are configured with the location of the proxy and connect directly to the proxy.

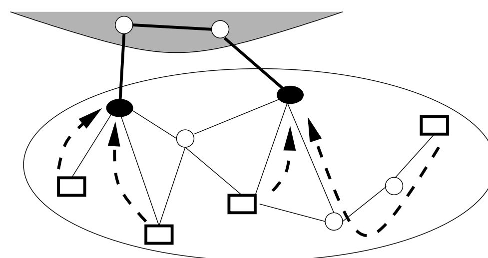


Figure 7: *Riptide Architecture:* Browser proxies use the OceanStore DOLR to automatically locate a nearby cache manager.

Finally, Riptide is tolerant to network and node failures. The underlying Tapestry DOLR automatically routes around failures in the network. If a node providing a certain service should fail, Tapestry will transparently route requests to an alternative service provider.

4.4 Adaptive Performance Optimizations

Riptide leverages the flexible, self-organizing mechanisms of OceanStore to improve the quality of service for clients. In this section, we describe how information is gathered and communicated among Riptide components to allow the cache managers to improve the quality of service for clients in their local area.

In typical proxy caching schemes, as shown in Fig-

ure 6, each client is statically configured to forward all requests to a specific proxy. The proxy is able to support cooperative caching among the client population because it is a shared resource that can aggregate the requests of many clients. In our scheme, each client has its own proxy and so the proxy is not well-suited to perform cooperative caching. Instead, the browser proxies send information to a nearby cache manager, as shown in Figure 7, by sending the messages through the OceanStore DOLR to the well-known identifier for a cache manager. Because of the locality properties of the DOLR, this message will find a nearby cache manager. Furthermore, if a cache manager fails, the OceanStore DOLR will transparently route the information to the next closest manager.

The proxy can send information of various forms to the cache managers. Most simply, a client can send a message notifying the manager that it was unable to access a document from the cache. Alternatively, the proxy may send a message indicating that it was able to successfully retrieve a document from the cache but that the retrieval cost was greater than expected.

The cache manager takes various actions depending on the content of the message. Information that a document was not available in the cache causes the cache manager to send a message to a nearby gateway requesting that the document be brought into the cache. To send the request to a nearby gateway, the manager simply sends a message through the OceanStore DOLR addressed to the well-known identifier of gateway services. Figure 4 shows the how a document might be inserted in the cache.

When a cache manager receives a hint that the cost, measured by latency or the number of network hops, to access content from the cache was too high, it may elect to create new replicas of content in the local area. Figure 8 shows how a cache manager can use documents cached by other managers to improve the service for its own client population. Creating a new replica of a cache document is a relatively inexpensive operation compared to cost of initially caching the document because creating a replica does require the expensive cryptographic operations that are needed to write an object into OceanStore. Thus, a cache manager creating a new replica is able to leverage the work performed by other managers and gateways in the infrastructure. This is how Riptide is able to provide the benefits of cooperative caching.

The cache manager may also improve the level of service available to its local client population by distributing additional replicas of popular content throughout the local area. Figure 9 shows how a cache manager can reduce the load of responding to requests for popular documents by spawning more replicas. With more replicas, clients can benefit from the aggregate read bandwidth all nodes hosting replicas. Additional replicas could also improve the latency of accessing popular documents from the cache by reducing the queueing delays at nodes hosting replicas. Note, because content is located directly through Tapestry, replicating content does not require any additional directory update steps.

4.5 Push Caching

We have thus far described Riptide as a system that performs a variation of proxy caching on top of OceanStore. We now describe how the same components can be used to support push-caching. While proxy caching is a passive technique that responds to streams of user requests, push-caching is an active technique that proactively pushes new content throughout the network so that it is nearby and current when requested by a client.

To provide push-caching capabilities, Riptide relies on OceanStore's dissemination tree discussed in Section 3.3. As cache managers create replicas of content, the replicas automatically tie themselves in to the document-specific dissemination tree. As the gateways update the content stored in the cache, the updates are pushed down the dissemination tree to all of the replicas.

Riptide can use any one of several techniques to determine when documents are changed by the content provider. Most simply, Riptide can rely on the browser proxies to observe that cached content is no longer valid, as defined by HTTP's cache control headers. The browser proxy would then send a hint to the cache manager; the cache manager would request that the gateway update the cached copy. The update would then be distributed to all other repli-

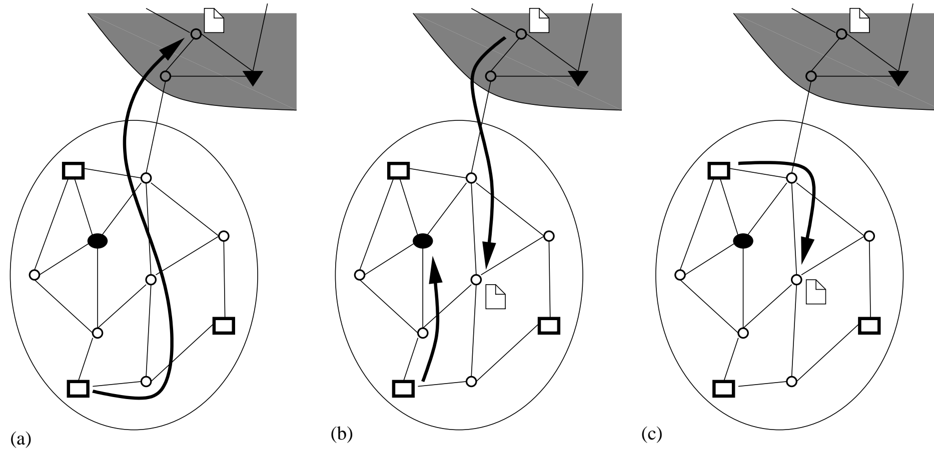


Figure 8: *Moving documents in the cache.* (a) The proxy successfully retrieves a document from the cache, but the retrieval cost was higher than expected. (b) The proxy notifies a nearby cache manager that retrieving the document from the cache is too costly. The cache manager creates a replica of the document and places it close to its local user population. (c) Another proxy requests the same document and finds it rapidly.

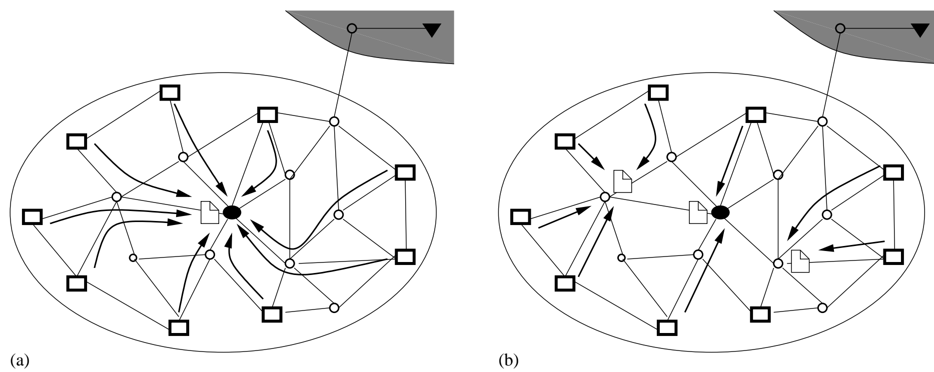


Figure 9: *Replicating documents in the cache:* (a) The load for serving a popular document from the cache can be excessive. (b) By replicating the content, the load on each individual replica is reduced.

cas. If the gateway finds that the content has not changed, it can push a heartbeat down the dissemination tree. A heartbeat is a small certificate used by the underlying system to indicate a version created some time ago is still current. When it receives a heartbeat, a replica knows that it is still connected to the dissemination tree and will receive any updates to the object.

Alternatively, if the gateway observes that its clients are frequently requesting that certain documents be updated, it can install a monitor to periodically poll the origin server for the content. By properly adjusting the polling frequency to account for the cache expiration policies of a document, a monitor watching a popular document can ensure that all replicas are current while reducing the total load on the origin server.

Finally, if a content provider is willing to cooperate actively with a gateway, it can proactively push new content to the gateway and in to the cache. To allow clients to verify that content was indeed current, the gateway would push heartbeats down the dissemination tree.

In Section 6, we will present an evaluation of monitor-based pushed caching. Introspective push caching and publisher-assisted push caching techniques are not examined any further.

The effectiveness of Riptide's push-caching capabilities are intimately related to the efficiency of the dissemination trees constructed by the underlying system. An optimal dissemination tree would use one interdomain link to access each stub and would cross that link exactly one time. Trees with minimal interdomain crossings are preferable because they minimize the amount of traffic sent over the interdomain links that tend to be congested, costly, or of limited bandwidth. To creating efficient trees, the system depends on the locality properties of the underlying DOLR. Figure 10 shows examples of poorly-constructed and well-constructed dissemination trees.

Previous research has shown that using a multicast tree for the purpose of push-caching can be an effective technique for popular documents that change frequently [16, 32]. We will show in Section 6 that Riptide can shield load from the origin web server and reduce the bandwidth requirements over the interdomain link. Further, we will show that push-caching via the dissemination tree can propagate content updates with sufficient efficiency to allow content that is traditionally marked uncacheable to be cached in Riptide.

5 Implementation

In this section, we describe the current implementation of the Riptide web cache. We have implemented a prototype of the Riptide caching system. It runs on the OceanStore [23] distributed storage infrastructure which in turn uses Tapestry [50] and attenuated Bloom filters [29] for location and routing. Riptide, like OceanStore is written completely in Java using the SandStorm staged, event-driven architecture [42]. We use the NBIO [43] library to provide an asynchronous, event-based Java interface to the network. The implementation of Riptide contains 2200 semicolons. The implementation of an auxiliary asynchronous HTTP library described in Section 5.2 contains approximately 1200 semicolons.

5.1 Replica Placement Strategies

The effectiveness of the cache manager depends greatly on how efficiently it can place replicas of content in the local area so that the browser proxies can find and retrieve them quickly. The cache manager relies on the primitives provided by OceanStore for flexibility in replica placement. Prior to the implementation of Riptide, OceanStore provided primitives for only one type replica placement. That primitive simply allowed a client to place a replica on its own node. Using only that primitive, Riptide would be forced to cache all content at the nodes hosting cache manager services. This would limit the scalability and fault-tolerance of the design. In order to realize the full flexibility of our design, we first developed another replica placement primitive.

To understand the new primitive, one must first understand how Tapestry locates objects in the infrastructure. We will refer to Figure 11(a) throughout the description. Tapestry uses its distributed in-

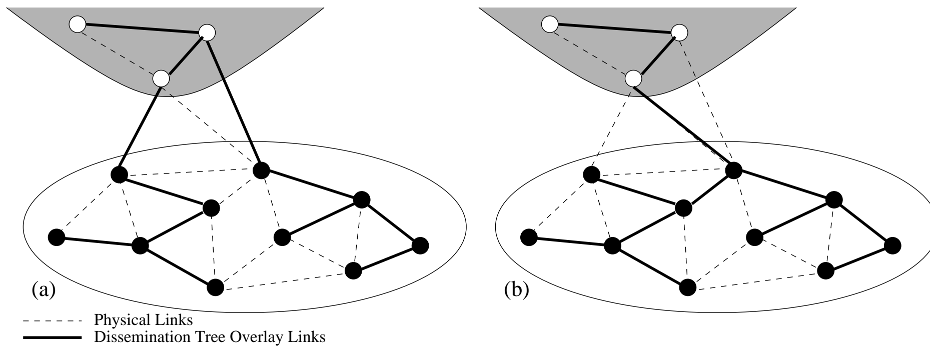


Figure 10: *Effective Dissemination Tree Building.* (a) Dissemination trees that use multiple interdomain links (or cross a single interdomain link multiple times) are less efficient. (b) An optimal dissemination tree crosses exactly one interdomain link for each stub that contains a replica.

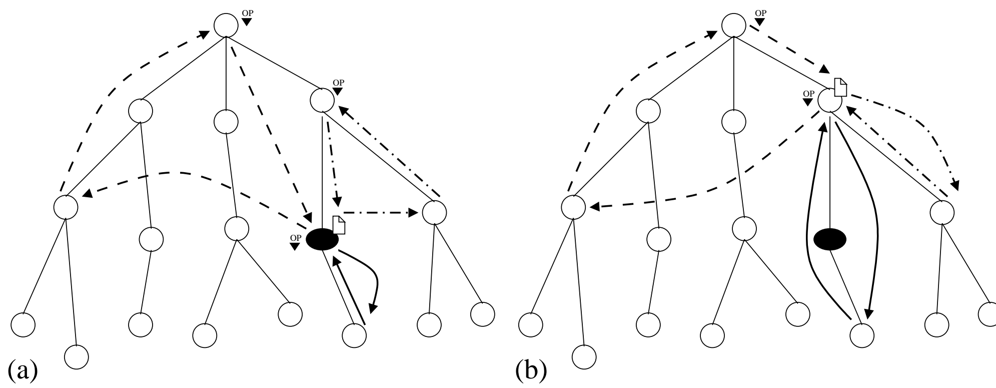


Figure 11: *Replica Placement Strategies:* The figure shows the Tapestry mesh arranged as a tree rooted at the given object's root nodes. (a) The cache manager uses the default replica placement primitive to create a replica on the local node. The steps taken to locate the replica are shown for three nodes. (a) The cache manager uses the new replica placement primitive to create a replica one hop closer to the object's root. Again, the steps taken to locate the replica are shown for three nodes.

dex to arrange all nodes into a *per-object* tree rooted at the node whose identifier most closely matches the object's GUID. Tapestry's index allows it to maintain all trees for all objects simultaneously. During the publish operation, Tapestry traverses all nodes between the object being published and the root of that object's tree. During the tree traversal, Tapestry updates its distributed index and deposits a *object-pointer* to the location of the object. The object-pointer may refer to the current node. Figure 11 shows which nodes would store object-pointers for the given object.

Object location proceeds in three steps. First, Tapestry climbs toward the root of the tree until it finds a object-pointer. In the worst case, Tapestry climbs the tree all the way to the root. If the object is published in the infrastructure, the root will contain a object-pointer to the object. If the root does not contain a object-pointer to the object, Tapestry deduces that the object does not exist and sends a failure response back to the client. Assuming that a object-pointer was found, in the second step, Tapestry follows the object-pointer to the requested object. Finally, the object is returned directly to the requesting client. Figure 11(a) shows the location process for three nodes in the infrastructure. Note, for nodes contained in the tree rooted at the node storing the object, the object-pointer found will refer to the node performing the query. In this case, the second step requires no additional network messages. In such cases, the second step is not shown in the figure.

The new replica placement primitive that we implemented takes advantage of Tapestry's object location algorithm to place replicas on nodes where they can be found by the more clients in the local area more quickly. The primitive works by partially climbing the tree towards the object's root node. After several hops toward the root, the request halts its climb and creates a replica of the object. Figure 11(b) shows an example where the cache manager has used the new primitive to create a replica of the object one hop up the object's tree. The figure also shows the three steps in the procedure for locating this new replica from three locations in the network. By climbing the tree several hops before creating a replica, the number of nodes contained in the sub-tree rooted at the replica increases. For some of those clients, locating the object now requires an extra network hop. Because of Tapestry's locality properties, this extra hop will usually be short and fast. For a much greater number of clients, locating the object will proceed more directly without the cost of the location procedure's second step.

We will show in Section 6.5 how this new replica placement primitive can be used to distribute replicas throughout the network.

5.2 Non-Blocking, Event-Driven HTTP Library

While the Java specification does provide several utility classes for applications to access web content, the Java standard library lacks an implementation of a full HTTP library. So, before implementing Riptide, we first developed a Java HTTP library that utilizes the non-blocking network interface provided by NBIO.

The main interfaces to the library are the `HttpClient` and `HttpServer` classes. An application that needs to retrieve web content uses the `HttpClient`. The application creates an `HttpRequest` message and enqueues it on the sink of the `HttpClient`. The `HttpClient` makes a connection to the origin server, fetches the document, and then places the response back on the application's sink.

Applications that need to service requests for web content use the `HttpServer`. The `HttpServer` opens a server socket on a specified port listening for incoming connections. After receiving a request from the network, the `HttpServer` posts the `HttpRequest` to the application's sink. The application constructs an `HttpResponse` object and enqueues it onto the `HttpServer`'s sink. The `HttpServer` then sends the response back to the requesting agent.

An application can create a simple proxy by employing both an `HttpClient` and `HttpServer`. The proxy simply passes requests from the server directly to the client and transfers responses directly from the client back to the server.

The library code is based loosely on code developed by Matt Welsh for the Haboob web server [42].

Both the `HttpClient` and `HttpServer` use the store-and-forward approach to message transmission. The library supports both HTTP/1.0 and HTTP/1.1 protocols. For HTTP/1.1, the library supports both identity and chunked transfer codings. The library provides support for persistent connections; it maintains network ordering in the library code so that application can handle requests and responses in any order. The library allows the application to flexibly control and query request and response header fields. The library allows the application to forward all requests or responses to other proxies, even those developed using other libraries. Finally, the library allows applications to tag requests and responses with arbitrary objects for easier event continuations.

5.3 Browser Proxy

The implementation of the browser proxy provides all of the functionality described in Section 4.2.2. The proxy, shown in Figure 12, uses components from the HTTP library to receive requests from the browser and retrieve content not available in the cache from the origin server. The browser proxy contains a small component to perform the relatively simple task of translating from HTTP requests to OceanStore read requests and from OceanStore read results to HTTP responses. The proxy uses TCP/IP for all HTTP traffic and the OceanStore DOLR for all other communication. The protocol for reading documents from the cache is defined by OceanStore; the format of the cache hints sent to the cache managers is defined by Riptide.

The current implementation of the browser proxy contains several notable simplifications. First, the browser proxy does not perform any cacheability checks based on the URL of the HTTP request. The proxy forwards all requests to the OceanStore infrastructure. Because these checks are merely simple string comparisons, we do not expect the inclusion of these checks to significantly impact the system. The simplification has no impact on our performance results because our current simulated workload based on SPECweb99 contains static content named by URLs of uniform form. The second simplification is that the browser proxy sends only one type of hint to the cache managers. The current implementation sends only cache miss hints as in the example shown in Figure 4. Our current simulation environment does not provide enough scale to consider the use of latency miss hints as described in the example of Figure 8.

5.4 Gateway

The implementation of the gateway provides the basic functionality described in Section 4.2.3. If the gateway receives a request to cache content that has never before been cache, the gateway will create a new data object in the OceanStore infrastructure. If the content has been previously cached, the gateway will simply update the data object with the new content. The gateway, shown in Figure 13, uses the `HttpClient` from the HTTP library to retrieve documents from the origin server. All other communication is performed using the OceanStore DOLR.

Currently, the gateway only caches content for which it receives a request to cache. It does not include any mechanisms, such as crawling, for proactively finding new content to cache. Adding functionality to crawl web pages when the gateway is under light load could decrease the number of compulsory cache misses suffered by clients.

The gateway does, however, allow for the creation of a number of small monitors to support push-caching, as described in Section 4.5. Monitors are timers with a small amount of state that defines what content they are monitoring and the frequency with which they are monitoring it. When the timer elapses, the monitor triggers the gateway to check that the cached copy of the content it is monitoring is still current. During the check, the gateway will either update the cache with new content or disseminate a heartbeat that verifies the old content is still current. By requesting that the gateway perform this check periodically, the monitor can ensure that content is actively pushed down the dissemination tree to replicas throughout the network. While a full implementation of the monitor logic would determine which sites to monitor introspectively based on its request stream, the current implementation monitors a statically defined set of URLs at a statically

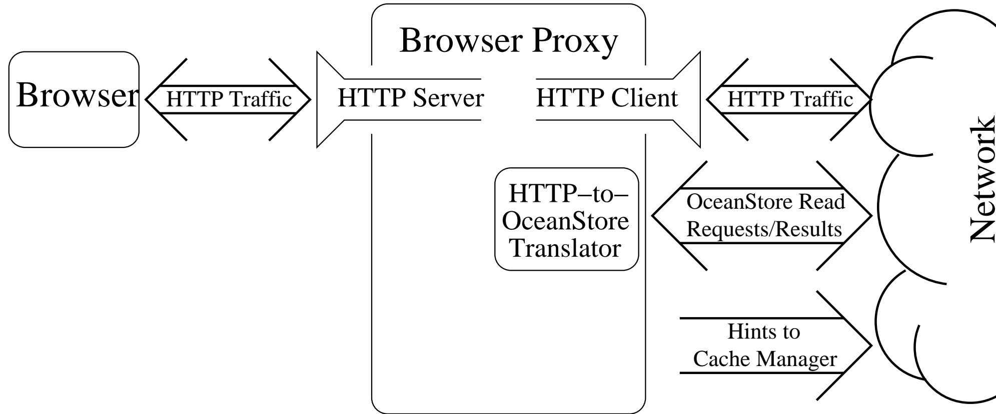


Figure 12: *Implementation of the Browser Proxy.* The browser proxy communicates with the browser using only standard HTTP. It communicates with the network using HTTP, OceanStore protocols, and application-level messages.

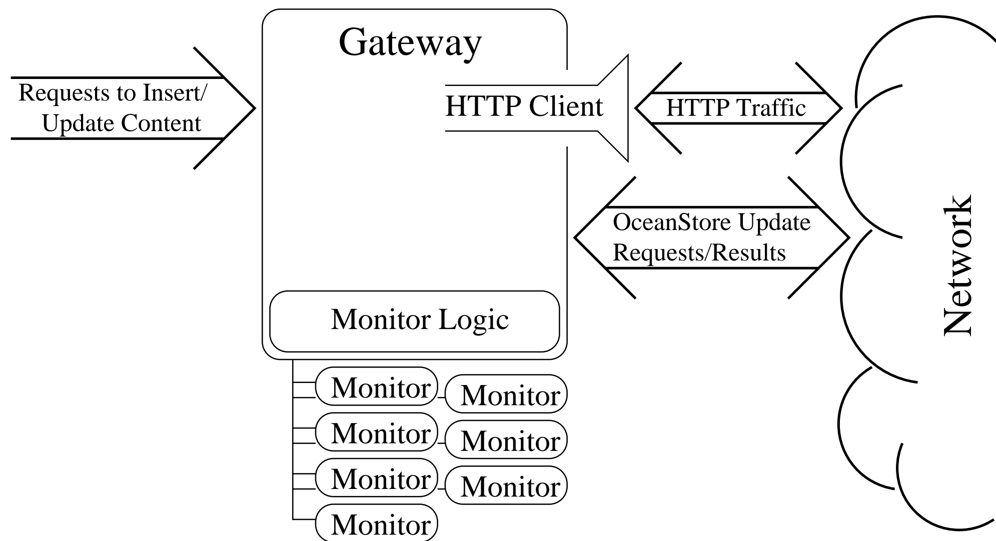


Figure 13: *Implementation of the Gateway.* The gateway accepts requests to insert or update documents in the cache. It retrieves the document from the origin server using standard HTTP and issues appropriate OceanStore commands to update the cache.

determined time interval.

Because gateways potentially serve large numbers of clients, there could be some concern that gateways are bottlenecks, limiting how quickly new content is brought into the cache. Note, however, that the work performed by the gateway is trivially parallelizable. The gateway performs the same sequence of steps for each document: it fetches the document from the origin server, compares the retrieved content with the cached content, and potentially updates the cached content. The work of servicing a request for one URL is completely independent of servicing a request for another URL. The gateway implementation could be trivially modified to run on a cluster or a number of nodes distributed throughout the infrastructure. One challenge to implementing a gateway to run on a number of nodes is ensuring that the nodes are not duplicating work. This could be solved for a cluster implementation by creating a front-end to filter requests to eliminate duplicates.

Finally, the gateway could reduce the amount of data it updates by describing content changes with a HTML differencing tool. Currently, if the gateway detects that the content of a page has changed, it submits an update to replace all of the old content with the new content. This results in updates that replace much more data than necessary. Using HTML differencing tools [9, 10] would reduce the size updates which would in turn reduce the load imposed by the gateway on inner rings. We have not attempted to integrated a differencing tool because the current OceanStore storage structure cannot efficiently handle the complicated updates it would produce. Work is currently underway to augment the storage representation with support for variable block sizes which would allow the use of more sophisticated updates. The unnecessarily large updates created by the system have not posed a problem in our evaluation because we have focussed on hit latencies and the effectiveness of the cache managers; we have not driven the gateways into overload.

5.5 Cache Manager

The cache manager, as described in Section 4.2.4, is the component that works greedily and introspectively to improve the quality of service for clients in the local area. By virtue of its responsibility, the design of the cache manager could be arbitrarily complex, implementing complicated predictive models or maintaining large amount of historical state. With our implementation of the cache manager, we have tried to keep the algorithms simple to focus on the architecture of the larger system. If the design proves to be effective, the cache manager could be extended to include the latest predictive caching techniques.

In the current implementation, the cache manager works greedily and without reference to any historical behavior. When it receives a hint from a browser proxy that some content could not be found locally, the cache manager attempts to create a replica in the local area. If it is unable to create a replica, it forwards a request to the nearest gateway to insert the document into the infrastructure.

The current implementation of the cache manager can use one of two different replica placement strategies. Most simply, the cache manager will cache content locally, creating replicas of content on the local node. As an example of a more advanced technique, the cache manager can also use the new replica placement primitive described in Section 5.1. Because Tapestry uses a different set of routing paths for each different object, using the advanced replica placement primitive results in a cloud of cached content centered about the cache manager. Considering other techniques for replica management is an area of ongoing research.

6 Results

In this section, we describe a number of experiments and their results performed to better understand the strengths and weaknesses of our design. First, we describe the experimental setup including how we simulate load and model the network. We start to explore the system by measuring the latency of the proxy. We continue by measuring the overhead of locating objects in Tapestry. We see that, unfortunately, the implementation of Tapestry does not exhibit the locality that its design promises. Next, we compare the document retrieval latency of Riptide to the la-

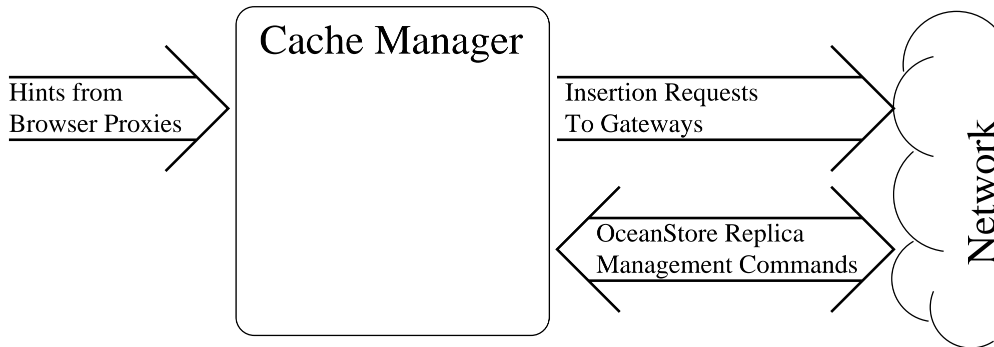


Figure 14: *Implementation of the Cache Manager.* The cache receives hints from nearby browser proxies. Depending on the type of hint, the cache manager forwards a request to a gateway or issues commands to the underlying system to direct the number and placement of replicas.

tency of a proxy cache. The goal of this comparison is not report an exact latency for each system; in fact, we do not have an infrastructure suitable for running such tests under truly realistic conditions. Instead, we hope to show with these measurements, that the Riptide web cache can deliver content to clients with reasonable latency. Next, we run experiments that demonstrate the real strengths of the Riptide architecture including automatic service and replica location and the ability to adapt to a changing network without client configuration changes. Finally, we present a preliminary exploration of how Riptide could be used to support push-caching.

6.1 Experimental Setup

To drive the system, we applied a simulated load generated by a modified version of the client load generator used in [42]. The traffic generator continually requests documents using a distribution specified by the SPECweb99 suite [8]. Between each request, the load generator inserts a fixed 10 ms of think time. The load generator uses a number of machines and threads to simulate many clients. The file set size is over 1.75 gigabytes, much larger than the available cache space available at any node.

We place an additional restriction on the SPECweb99 file set that only 80% of the documents are cacheable. This number reflects the average cacheability used in the Web Polygraph's web caching benchmark PolyMix-3 workload [33]. We also insert an average of 2.5 seconds of server think time when requesting a document from the origin server, as is done in the PolyMix-3 workload.

All experiments were run on top of simulated networks constructed using the *transit-stub* model of GT-ITM [48]. We augment the GT-ITM model with bandwidth numbers as follows. All stub-to-stub edges are 100 Mb/s, all stub-to-transit edges are 1.5 Mb/s, and all transit-to-transit edges are 45 Mb/s. These values were chosen to model Fast Ethernet, T1, and T3 connections respectively. Intra-stub link latencies varied from 1–7 ms; latencies for interdomain links were 10–30 ms.

Each graph created using the GT-ITM network model contained 1100 nodes, and each stub domain contained on average 100 nodes. The nodes performing OceanStore-specific roles were distributed across the (simulated) wide-area. To simulate a small organization, we placed 95 nodes serving various web caching roles in a single stub. Experimental results were similar all on generated networks; we present results from a single topology.

The experimental testbed used for all experiments consists of a local cluster of forty-two machines at Berkeley. Each machine in the cluster is a IBM xSeries 330 1U rackmount PC with dual 1.0 GHz Pentium III CPUs, 1.5 GB ECC PC133 SDRAM,

and two 36 GB IBM UltraStar 36LZX hard drives connected via gigabit Ethernet.

The web server hosting all of the content for the experiments was hosted on the management node of the cluster, reachable from the cluster in under 0.1 ms.

Finally, to compare Riptide against an existing, commonly-understood cache architecture, we constructed a simple proxy cache that implements LRU replacement algorithm. For experiments using the proxy cache, the cache is deployed in the stub domain with the clients, and the clients are configured to connect directly to the proxy.

6.1.1 Artifacts of the OceanStore DOLR

In many of the sections below, experimental results will be presented for two different DOLR configurations. In one configuration, we use only Tapestry for DOLR; in the other configuration, we use Tapestry augmented by attenuated Bloom filters for DOLR. We present results from both configurations for several reasons.

The current implementation of Tapestry should be considered preliminary. While the APIs are stable, a great deal of work remains to provide consistently good routing. In the current implementation, Tapestry especially has problems with locality. It frequently routes through other domains to find resources that are actually stored in the local domain. It also tends to take several hops to locate an object that is only one physical hop away. These are recognized problems by the Tapestry development team and the current re-implementation effort is focusing on the locality issues. The lack of locality negatively impacts the results by making objects and services appear much farther away than they really are.

To combat the locality issues of the current Tapestry implementation, we sometimes augment Tapestry with attenuated Bloom filters. The Bloom filter routing optimization is especially efficient at locating nearby replicas and services quickly. The results of the experiments that use the Bloom filters hint at the performance that could be attained as Tapestry becomes more efficient.

Size (bytes)	Direct (ms)	Proxy (ms)
1024	1.35	23.30
4096	1.70	24.07
10240	1.90	24.40
40960	5.25	25.35
102400	9.90	38.81
409600	36.00	63.35

Table 1: *Proxy Latency:* The proxy imposes an overhead of slightly more than 20 ms on requests for small objects. Much of this delay is due to a timeout in the NBIO library.

6.2 Proxy Latency

All requests for content pass through the browser proxy. Consequently, it is important that the overhead imposed by the proxy during protocol translation and secure hash computation be minimal. To measure the overhead of the proxy, we configured the browser proxy to act as a transparent proxy, a proxy that retransmits requests without modification. Specifically, when configured as a transparent proxy, the browser proxy will forward the request to the origin server without checking Riptide for cached content. We then measured the latency to retrieve content of variable size with a direct connection to the network and through the proxy.

Table 1 presents the average latency of retrieving documents through the proxy. Each data point is the average of latency of twenty requests. The overhead imposed by the proxy when requesting small object is, on average, slightly more than 20 ms. We have found that 20 ms of the latency is an artifact of a timeout used in the NBIO implementation of non-blocking networking sockets. As we decrease the timeout, the latency of the proxy drops correspondingly at a cost of increased CPU load. Balancing proxy latency with CPU load, we have left the latency imposed by the timeout at 20 ms for the remainder of the experiments. We hope this issue will be resolved with the widespread release of Java 1.4 with its native support for non-blocking I/O. Excepting the latency imposed by the non-blocking sockets implementation, the overhead of the proxy is minimal.

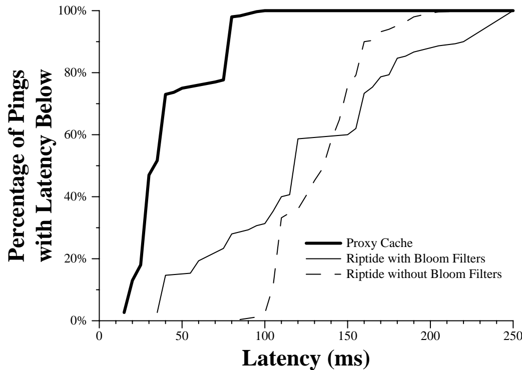


Figure 15: *Service Location Overhead:* We compare the latency to reach the proxy cache with the latency to reach the cache manager both with and without bloom filters. Using Tapestry to locate services is more costly than contacting the service directly.

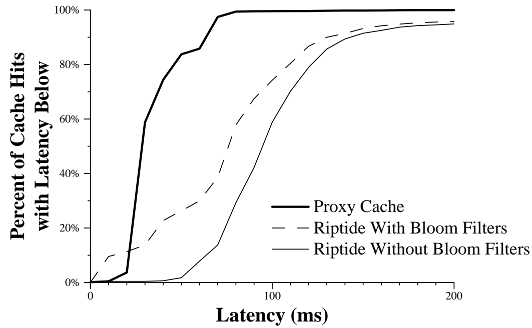


Figure 16: *Hit Latency:* A cumulative distribution function of the latencies to locate and retrieve a document from the web cache. Latencies are presented for the proxy cache and Riptide with and without bloom filters.

6.3 Service Location Overhead

Next, we wanted to examine the overhead of using Tapestry to locate services dynamically. While there are no service location operations in the critical path for reading a document from the cache, it is an interesting metric that hints at the overhead imposed by Tapestry in locating objects. To measure the overhead, each client would ping a service provider. When measuring the latency to the proxy cache, a client would contact the proxy cache specified by their configuration file. When measuring the latency to the cache manager, a client would use Tapestry to locate a nearby cache manager; the client is *not* configured to know the location of any cache manager.

Figure 15 shows the distribution of ping latencies. Not surprisingly, clients with foreknowledge of the location of their service provider can reach that provider faster than those that have no such foreknowledge. Clients can usually reach the nearby proxy cache in under 50 ms; routing through Tapestry, clients regularly see latencies up to 200 ms. Notice, that by using bloom filters to aid in service discovery, the latency to reach the service provider drops below 50 ms for some clients.

6.4 Cache Latency

Next, we compare the latency to retrieve documents from the Riptide web cache against the latency to retrieve documents from a traditional proxy cache. While the strength and emphasis of our design is adaptability and ease of maintenance and configuration and not low hit latency, it is important that it perform reasonably well. The experiment below is designed to provide a fair comparison with the proxy cache. Specifically, the proxy cache is not measured in an overload situation. Obviously, if driven with sufficient load, the distributed Riptide web cache could perform better than the single-node proxy cache. Our intent with this experiment is to measure the cost of retrieving content when neither system is suffering from under-provisioning.

Figure 16 displays the cumulative distribution functions (CDF) of the latency to retrieve documents from the proxy cache and Riptide. The cache manager uses the simple primitive to create replicas on the node hosting the cache manager. Using the proxy cache, 80% of hits are returned in under 50 ms. Clients using a proxy cache see low latencies because they know the location of the proxy cache and can route directly to the cache.

Using Riptide without Bloom filters, clients need to wait almost 120 ms to see 80% of the hits. Most of the difference in latency can be attributed to latency of

the extra hops required to locate documents through Tapestry.

Allowing Riptide to use bloom filters to aid in document location improves latency. Figure 16 shows that when using bloom filters, a number of objects can retrieved with latencies rivaling the proxy cache scheme.

The reader may notice that the CDFs for Riptide in Figure 16 do not reach 1.0. This is an artifact of implementing OceanStore and Riptide in the garbage-collected Java programming language. All current production Java Virtual Machines (JVMs) we have surveyed use so-called "stop the world" collectors, in which every thread in the system is halted while the garbage collector runs[1]. Any requests currently being processed when garbage collection starts are stalled for on the order of one hundred milliseconds. This penalty can be multiplied if a request is stalled by garbage collection on several nodes as it is routed through the infrastructure. The performance problems introduced by garbage collection are much more noticeable with Riptide than the proxy cache because Riptide is running on top of a much larger software stack that forces much more garbage collection.

6.5 Automatic Service Location and Load Balancing

One of the unique features of Riptide is its ability to automatically find and use new resources as they are added to the network. To demonstrate this feature, we performed several experiments.

First, we brought several cache managers on-line while the system was running and measured the number of requests each manager received over time. The system begins executing with one cache manager; a second cache manager is added after 5 minutes; and a third cache manager is installed after 10 minutes. Figure 17 shows the number of hints received by each cache manager during each time interval. As the system warms up, the first cache manager sees an increasing request rate. When the second cache manager comes on line during the 30th time interval, it

[1]We currently use JDK 1.3 for Linux from IBM. See http://www.ibm.com/developerworks/java/jdk/linux130/

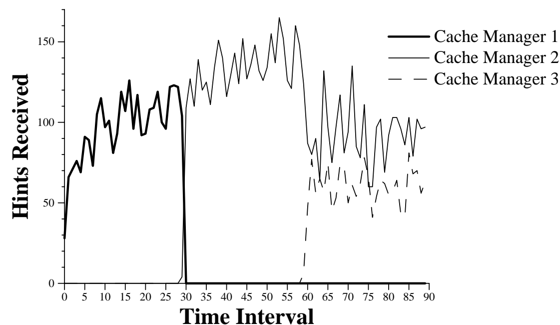


Figure 17: *Automatic Service Location:* This figure shows how requests to cache managers are rerouted as new services enter the network. By using Tapestry to locate services, the system automatically finds the closest provider, even as providers enter and leave the network. Each time interval is 10 seconds.

is closer to all of the clients then the first cache manager. Consequently, all of the traffic shifts to the second cache manager. When the third cache manager begins to provide service during the 60th time interval, it is closer to almost half of the clients and thus automatically begins to receive roughly half of the client traffic. It is important to note that as new cache managers are added to the system, no configuration changes are made anywhere in the system.

In the second experiment, we simulated a flash crowd, a sudden increase in requests, for a single document. In response to the flash crowd, the cache manager creates additional replicas of the popular content. We measure the number of requests that each replica receives. The system begins with a single copy of the document. After several minutes, a second replica is created; a third replica is created after several more minutes. Browser proxies, using Tapestry, discover the new replicas as they are created. Figure 18 shows the result of this experiment. Each replicas begins to receive and service requests as soon as it is created. That each replica does not receive an equal proportion of the requests is an artifact of the experimental setup. Because we simulate the flash-crowd using a small number of hosts, the load generators apply load from only five locations in the network. Because requests originate from only

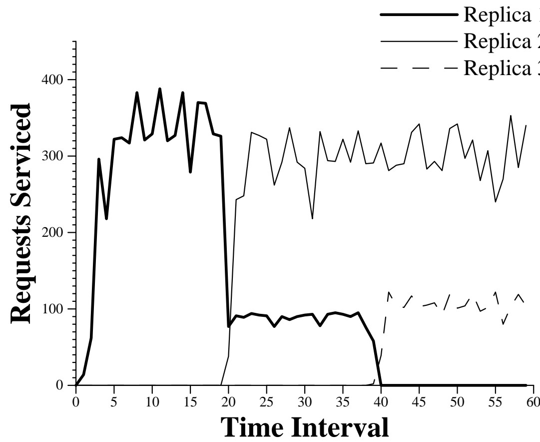


Figure 18: *Automatic Replica Location:* This figures shows how requests are routed to replicas as they enter the system. By using Tapestry, the system automatically discovers new replicas as they are created. Each time interval is 10 seconds.

several locations, Tapestry does not route requests to replicas uniformly.

Finally, cache managers also have the flexibility to execute a wide variety of powerful replica placement policies. To explore the effects of using a different policy, we modified the cache manager to use the advanced replica placement primitive discussed in Section 5.1. Because the primitive chooses the node for the new replica based on the document-specific routing tree, using the advanced primitive results in a cloud of cached content distributed among a number of nodes with the cache manager in the center of the cloud. Consequently, the load for serving content from the cache is distributed across the nodes in the cloud. Figure 19 shows how load can be distributed by using the advanced replica placement primitive. In the figure, nodes are sorted by the number of requests that they receive. Nodes receive a varying number of requests because they host replicas of varying popularity. The cache manager has spread the load of serving cache requests across a number of nodes in the infrastructure. This is a clear improvement over the simple alternative; that alternative would force a single node to serve all of the requests handled by the

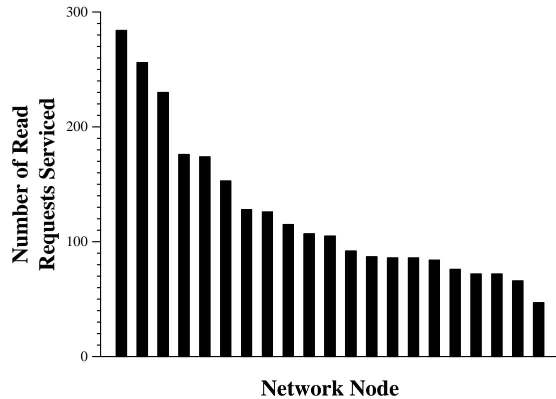


Figure 19: *Effectiveness of Replica Placement Strategy:* The histogram shows the number of requests satisfied by a number of nodes in the network over a 150 second period. The nodes on the x-axis are sorted by the number of requests received. The units on the x-axis are left unmarked because the exact identity of the nodes is irrelevant. By using the advanced replica placement primitive, the cache manager has spread the load of serving cache requests across a number of nodes in the infrastructure. This is a clear improvement over the simple alternative; that alternative would force a single node to serve all of the requests handled by the collection of nodes shown in the figure.

collection of nodes shown in the figure.

6.6 Viability of Push-Caching

To better understand the viability of push-caching, we monitored the main page of several popular content providers. We fetched the current content every two minutes and compared it against the previous version of the page. If it had been updated, we stored the page. We collected the content updates for 10 popular sites from July 18, 2002 to July 30, 2002. The sites profiled served general news, financial news, sports news, or tech-related information.

We analyzed the data to determine how much of the page was updated and how frequently. The results of this analysis, which we call the *update profile*, appear in Figures 20 and 21. The top curve in each update profile shows the size of the document in bytes. The bottom curve shows the time intervals during which the content was change and how many bytes of content were changed. To compute the number of bytes changed, we simply used Unix's `diff` tool. Because `diff` uses a line-based comparison, this computation is somewhat conservative. Using tools from current research on HTML diffing [1, 24] would provide more accurate measurements.

Figure 20 presents the update profile of several sites that appear to be good candidates for push-caching. These sites update their content periodically and do not modify the metadata or HTML code between content updates. This means that the content, once pushed down the dissemination tree to the replicas, can be used to serve a number of requests. Furthermore, the number of bytes changed when the content is updated is small. This means that only a small diff needs to be pushed down the dissemination tree to update the replicas.

Figure 21 presents the update profile typical of the many sites that may not benefit from push-caching. This update profile shows that the content of the document had changed *every* time the document was checked. Unfortunately, push-caching is not effective if the page changes every time it is accessed. However, a manual inspection of the changes that occur on reload reveal that the content that changes is related to advertising or other auxiliary content. This indicates, encouragingly, that if the sites that exhibit continuous document change used other models, such as those used by the sites profiled in Figure 20, then they too could be good candidates for push-caching.

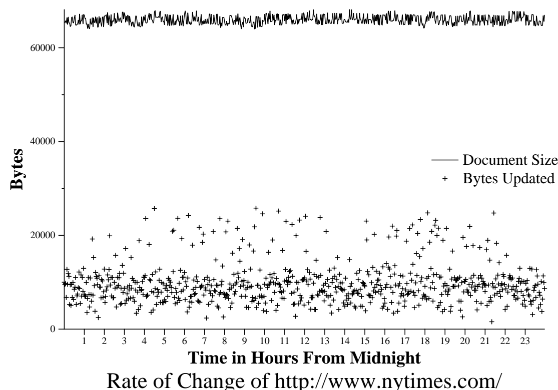


Figure 21: *Update Profile of http://www.nytimes.com/:* The update profile of this site shows that content is changed every time the page is requested. Manual inspection shows that the changes are usually related to advertising and other auxiliary content.

6.7 Push-Caching to Reduce Interdomain Bandwidth

We previously described how the use of OceanStore's dissemination tree can reduce the bandwidth con-

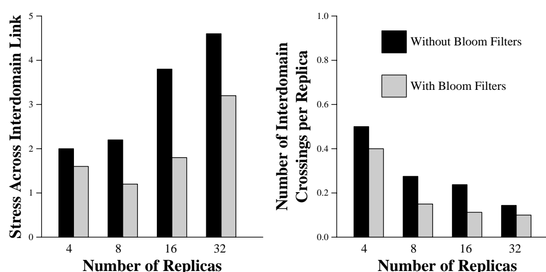


Figure 22: *Reducing Interdomain Bandwidth with the Dissemination Tree:* Binding replicas together in a dissemination tree reduces the amount of interdomain bandwidth required.

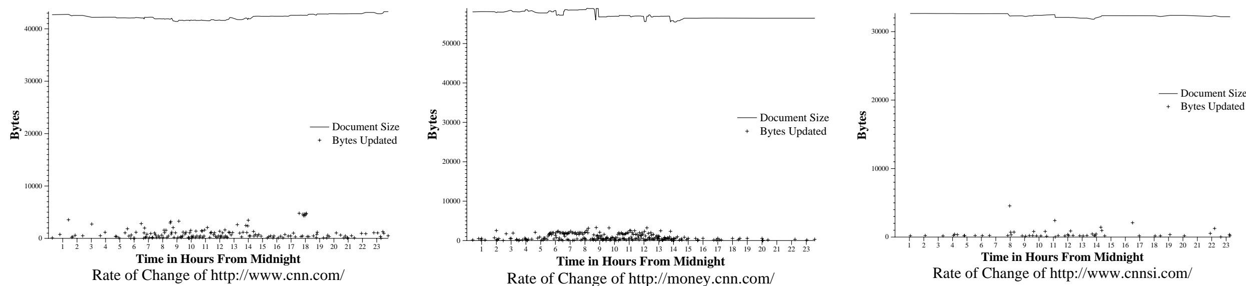


Figure 20: *Update Profile of Several Sites:* These update profiles show that content is updated only periodically, which allows for content to be shared, and that only a small portion of the content is updated at any time, which means that diffs required to update replicas are small compared to the document size. Notice that the financial site shows a much greater rate of change when the markets are open.

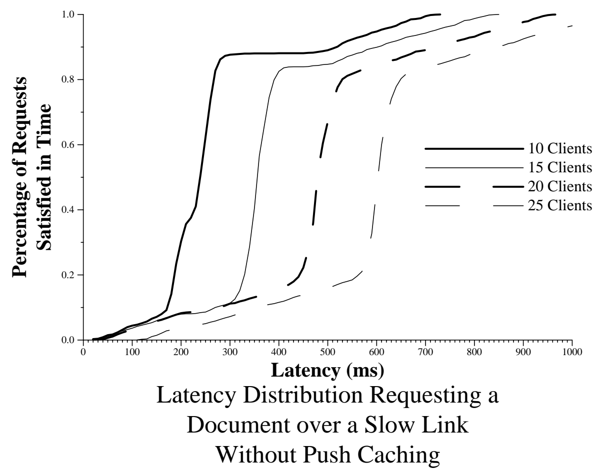
sumed on interdomain links by pushing updates to copies of content. Figure 22 shows the number of interdomain crossings used in the creation of a dissemination tree among a variable number of replicas. The graphs show that the dissemination tree is often able to save bandwidth by serving multiple replicas with a single interdomain crossing. If clients are accessing documents cached in a number of replicas bound together in a dissemination tree, updates can be pushed to these clients with many fewer domain crossings than if each client was required to contact the origin server directly. Results show that the dissemination tree becomes more efficient as the number of replicas increases. The tree does, however, cross the interdomain links more times than necessary. The dissemination tree should be better able to optimize the number interdomain crossing as Tapestry is better able to locate local resources.

6.8 Push-Caching to Reduce Server Load

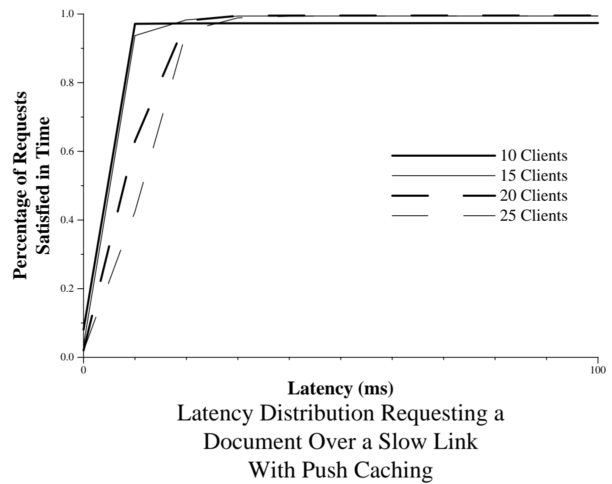
To study the effects of push-caching on server load and client-perceived latency, we built a simple web server that could simulate restricted bandwidth on its outbound link. A variable number of clients request the document from the server. In the base configuration, no caching is performed. Consequently, all requests must be serviced by the origin web server. As the number of clients and the number of requests increases, the origin's servers network link becomes saturated and the response latency increases reflecting queueing delays over the constrained link. In the optimized configuration, Riptide is used in the push-caching mode. Monitors are installed at the gateway to monitor the content hosted by the web server. When the gateway observes that content has changed, it updates the content in the cache and pushes that update to all other replicas. Clients first check the cache for a current copy of the document. If the gateway is performing well, clients will find an up-to-date copy of the document in the cache. If the client could not find a current copy of the document, it retrieves the document from the server.

For the results presented below, the document is considered stale after 5 seconds; the content of the document actually changes every 30 seconds. The downstream bandwidth of the web server is limited to 10 Mbit/s. The size of the document is 16 KB.

Figure 23 shows the CDF of the client-perceived latency for requesting documents from the bandwidth-limited server. Without push-caching, all requests must go to the origin server and responses are delayed by queueing effects at the constrained link. As clients issue more simultaneous requests, the effects of queueing on latency increase. With push-caching, most of the requests are satisfied by local replicas with little latency. When a replica is stale and un-



Latency Distribution Requesting a
Document over a Slow Link
Without Push Caching



Latency Distribution Requesting a
Document Over a Slow Link
With Push Caching

Figure 23: *Request Latency Over a Slow Link:* The graph on the left shows the latency when retrieving documents from the server in the system without push-caching. Once the link is saturated, the response time increases linearly with the number of clients. The graph on the right shows the latency when retrieving documents from a server in the system using Riptide's push-caching functionality. The latency of retrieving the document does not increase because updated are pushed down the dissemination tree to the replicas. Note the scale x-axis is 10 times smaller on the graph for the system using push-caching.

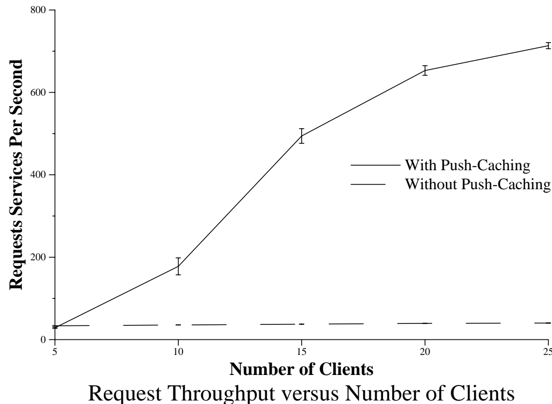


Figure 24: *Request Throughput:* The system using push-caching is able to satisfy many more requests per second than the base system. The graph shows the average and standard deviation of 200 seconds of simulation.

able to satisfy the request, the system forwards the request to the origin server. However, because most traffic is absorbed by the caches, the latency of the response from the origin server is much lower. Furthermore, the results are pushed to the other replicas in the system to serve client requests.

In addition to serving requests much faster, push-caching enables the system to service many more requests per second. Without push-caching, every request consumes some bandwidth over the network link that is the constrained resource in this system. With push-caching enabled, the bandwidth to service most requests comes from the relatively bandwidth-unconstrained stub domain. Only a few requests consume bandwidth over the interdomain link. By shifting the consumption of resources into the local area, the throughput of the system using push caching is much hight, as shown in Figure 24.

7 Related Work

There has been a great deal of research on web caching. Surveys by Wang [40] and Barish and Obraczka [2] show the number and variety of ideas explored. Hierarchical caching architectures became popular with the introduction of the Harvest [5] and Squid [44] proxy caches. Another example of hierarchical caching is the Adaptive Web Caching [49] system.

Attempts to distribute load among web caches resulted in a number of protocols to allow caches to query known neighbor caches in the hierarchy directly, without disturbing the parent. The Internet Cache Protocol (ICP) [46, 45] was the first proposal and used small UDP messages to query neighboring caches. If ICP cannot locate a neighboring cache willing to serve the content, it propagates the request up the hierarchy.

Another group of projects attempted to tie together small sets of caches at the institutional level. The Cache Array Routing Protocol (CARP) [39] uses a hashing function to route requests directly to one of a number of loosely coupled proxies. The CRISP web caching service [14] proposed to use a centralized directory for each cluster of proxies. A user's proxy would contact the directory to find of copy of the cached content. The Cachemesh project [41] proposed to use communication among proxies to develop cache placement and routing strategies. Cache Digests [34] and the Summary Cache [12] use Bloom filters to store a summary of the directory at each node. Proxies consult the bloom filters stored locally to find another proxy which (with high probability) is caching the document. Proxies periodically exchange information to update the Bloom filters. Note that in all of these proposals, there is a certain level of administrative coordination required to create clusters of proxies to participate in the schemes and there is some overhead required to keep the directories or hash functions current.

The Squirrel web cache [18] represents an interesting point in the design space. Built on top of the PASTRY [36] routing infrastructure, it builds a peer-to-peer caching network. The resources used for storing and delivering data come from client machines participating in the cache.

Akamai [20] has developed a Web cache network using the ideas of consistent hashing. The network distributes the load of serving content to prevent hot spots. The properties of consistent hashing allow the network to scale gracefully and use resources ef-

ficiently.

Finally, there have been several attempts to help clients locate a proxy server and configure a browser to use the proxy [15, 7]. As far as we know, none of these proposals have become standards.

8 Future Work

Several aspects of Riptide deserve further study. The work presented in this paper has only begun to look at the advantages in bandwidth and latency of pushing content changes to replicas. We would also like to examine additional replica management strategies for cache managers. Of particular interest is strategies that allow a single manager to detect "hot" documents and to create many replicas for a single document. Further, we would like to study the architecture as it is deployed across a wider simulated area; specifically, we would like to simulate the architecture across multiple stubs to simulate cross-organization cooperative caching. Finally, OceanStore has provisions for *versioning*, namely keeping every version of every document. In Riptide, this will enable Web browsers to perform *time travel* – clients will be able to look at the web at any point in the past, starting from the point that a gateway first imported the document into OceanStore.

9 Conclusion

In this paper, we present the Riptide distributed web caching architecture. Riptide is constructed on top of OceanStore and exploits mechanisms for decentralized object location and for pushing updates to replicas. This architecture requires minimal configuration, recovers transparently from network and server failures, and can be scaled as necessary to meet client demands. We measured the performance of a functional prototype on a simulated workload. We showed that the system is able to adapt as new services are added to the network and to distribute load across nodes in the network. Riptide incurs somewhat increased latency over a simple proxy-based architecture, but this increase in latency is modest and more than offset by the ability of the system to manage itself and adapt to adverse server loads.

References

[1] Gaurav Banga, Fred Douglis, and Michael Rabinovich. Optimistic deltas for WWW latency reduction. In *USENIX Technical Conference*, pages 289–303, 1997.

[2] Greg Barish and Katia Obraczka. World wide web caching: Trends and techniques, 2000.

[3] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, volume 13(7), pages 422–426, July 1970.

[4] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich. Web proxy caching: the devil is in the details, 1998.

[5] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical internet object cache. In *USENIX Annual Technical Conference*, pages 153–164, 1996.

[6] Yan Chen, Randy H. Katz, and John D. Kubitowicz. SCAN: A dynamic, scalable, and efficient content distribution network. In *Proceedings of the International Conference on Pervasive Computing*, August 2002.

[7] Netscape Corporation. Navigator proxy auto-config file format, http://wp.netscape.com/eng/mozilla/2.0/relnotes/demo/proxy-live.html.

[8] Standard Performance Evaluation Corporation. Specweb99, http://www.specbench.org/osg/web99/.

[9] Fred Douglis and Thomas Ball. Tracking and viewing changes on the web. In *USENIX Annual Technical Conference*, pages 165–176, 1996.

[10] Fred Douglis, Thomas Ball, Yih-Farn Chen, and Eleftherios Koutsofios. The ATT internet difference engine: Tracking and viewing changes on the web. *World Wide Web*, 1(1):27–44, 1998.

[11] Bradley M. Duska, David Marwood, and Michael J. Feeley. The measured access characteristics of world wide web client proxy caches. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[12] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking*, 8(3):281–293, 2000.

[13] Li Fan, Pei Cao, Wei Lin, and Quinn Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Measurement and Modeling of Computer Systems*, pages 178–187, 1999.

[14] S. Gadde, M. Rabinovich, and J. Chase. Reduce, reuse, recycle: an approach to building large internet caches. In *The Sixth Workshop on Hot Topics in Operationg Systems*, pages 93–98, 1997.

[15] Paul Gauthier, Josh Cohen, Martin Dunsmuir, and Charles Perkins. Web proxy auto-discovery protocol, http://www.web-cache.com/Writings/Internet-Drafts/draft-ietf-wrec-wpad-01.txt.

[16] James Gwertzman and Margo Seltzer. An analysis of geographical push-caching. 1997.

[17] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed data location in a dynamic network. In *Proc. of ACM SPAA*, 2002.

[18] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache.

[19] Kirk L. Johnson, John F. Carr, Mark S. Day, and M. Frans Kaashoek. The measured performance of content distribution networks. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, 2000.

[20] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin, and R. Panigrahy. Consistent hashing and random trees. distributed caching protocols for relieving hot spots on the world wide web. In *Proc. of ACM STOC*, May 1997.

[21] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. On the use and performance of content distribution networks. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, 2001.

[22] Tom M. Kroeger, Darrell D. E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[23] J. Kubiatowicz et al. Oceanstore: An architecture for global-scale persistent storage. In *Proc. of ASPLOS*. ACM, 2000.

[24] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta encoding and data compression for HTTP. In *SIGCOMM*, pages 181–194, 1997.

[25] NIST. FIPS 180-1 secure hash standard. April 1995.

[26] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve World-Wide Web latency. In *Proceedings of the ACM SIGCOMM '96 Conference*, Stanford University, CA, 1996.

[27] D. Povey and J. Harrison. A distributed internet cache. In *20th Australian Computer Science Conference*, 1997.

[28] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of SIGCOMM*. ACM, August 2001.

[29] S. Rhea and J. Kubiatowicz. Probabilistic location and routing. In *Proc. of INFOCOM*. IEEE, June 2002.

[30] S. Rhea, C. Wells, P. Eaton, D. Geels, B. Zhao, H. Weatherspoon, and J. Kubiatowicz. Maintenance free global storage in oceanstore. In *Proc. of IEEE Internet Computing*. IEEE, September 2001.

[31] P. Rodriguez, C. Spanner, and E. Biersack. Web caching architectures: hierarchical and distributed caching, 1999.

[32] Pablo Rodriguez and Ernst Biersack. Continuous multicast push of web documents over the internet. *IEEE Network Magazine*, 12(2):18–31, March 1998.

[33] Alex Rousskov, Matthew Weaver, and Duane Wessels. Polymix-3 workload, http://www.web-polygraph.org/docs/workloads/polymix-3/.

[34] Alex Rousskov and Duane Wessels. Cache digests. *Computer Networks and ISDN Systems*, 30(22-23):2155–2168, 1998.

[35] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large scale peer-to-peer systems. In *Proc. of IFIP/ACM Middleware*, November 2001.

[36] Antony Rowstron and Peter Druschel. Pastry: scalable, distribute object location and routing for large-sclale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, pages 329–350, November 2001.

[37] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of SIGCOMM*. ACM, August 2001.

[38] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonatoan S. Kehy. Design considerations for distributed caching on the internet. In *International Conference on Distributed Computing Systems*, pages 273–284, 1999.

[39] V. Valloppillil and K. Ross. Cache array routing protocol (carp), internet draft.

[40] Jia Wang. A survey of web caching schemes for the internet, 1999.

[41] Z. Wang and J. Crowcroft. Cachemesh: a distributed cache system for the world wide web. In *Web Cache Workshop*, 1997.

[42] M. Welsh, D. Culler, and E. Brewer. SEDA: An architecture for well-conditioned, scalable internet services. In *Proc. of ACM SOSP*, October 2001.

[43] Matt D. Welsh. Nbio: Non-blocking I/O for Java, http://www.cs.berkeley.edu/ mdw/proj/java-nbio/index.html.

[44] D. Wessels. The squid internet object cache http://squid.nlanr.net/Squid/.

[45] D. Wessels and K. Claffy. Application of internet cache protocol (ic), version 2, rfc 2187.

[46] D. Wessels and K. Claffy. Internet cache protocol (icp), version 2, rfc 2186.

[47] Duane Wessels and K Claffy. Evolution of the nlanr cache hierarchy: Global configuration challenges http://www.nlanr.net/Papers/Cache96/.

[48] E. Zegura, K. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Proc. of INFOCOM*, 1996.

[49] Lixia Zhang, Scott Michel, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson. Adaptive web caching: Towards a new global caching architecture. In *3rd International WWW Caching Workshop*, June 1998.

[50] B. Zhao, A. Joseph, and J. Kubiatowicz. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB//CSD-01-1141, U. C. Berkeley, 2001.