# Using Bundles for Web Content Delivery[*]

Craig E. Wills
Gregory Trott
Mikhail Mikhailov

Computer Science Department
Worcester Polytechnic Institute
Worcester, MA 01609

**Abstract**

The protocols used by the majority of Web transactions are HTTP/1.0 and HTTP/1.1. HTTP/1.0 is typically used with multiple concurrent connections between client and server during the process of Web page retrieval. This approach is inefficient because of the overhead of setting up and tearing down many TCP connections and because of the load imposed on servers and routers. HTTP/1.1 attempts to solve these problems through the use of persistent connections and pipelined requests, but there is inconsistent support for persistent connections, particularly with pipelining, from Web servers, user agents, and intermediaries. In addition, the use of persistent connections in HTTP/1.1 creates the problem of non-deterministic connection duration. Web browsers continue to open multiple concurrent TCP connections to the same server.

This paper examines the idea of packaging the set of objects embedded on a Web page into a single *bundle* object for retrieval by clients. Based on measurements from popular Web sites and an implementation of the bundle mechanism, we show that if embedded objects on a Web page are delivered to clients as a single bundle, the response time experienced by clients is better than that provided by currently deployed mechanisms. Our results indicate that the use of bundles provides shorter overall download times and reduced average object delays as compared to HTTP/1.0 and HTTP/1.1. This approach also reduces the load on the network and servers. Implementation of the mechanism requires no changes to the HTTP protocol.

**Keywords**: Web Performance, Persistent Connections, Content Delivery.

[*]This work extends work previously published in the Tenth International World Wide Web Conference, Hong Kong, May 2001. This work is partially supported by the National Science Foundation Grant CCR-9988250

## 1 Introduction

The World Wide Web has significantly evolved over the past few years. The amount of offered content and the number of content consumers have grown exponentially, complexity and richness of content has increased, functionality and performance of Web servers and user agents has improved. Web transactions, carried over the HyperText Transfer Protocol (HTTP) [2, 7] running on top of the Transmission Control Protocol (TCP) [22], account for 70-75% of traffic on the Internet backbone according to one study [4]. With the Web being the main application on the Internet, it is vital to ensure the efficient use of network (and server) resources by Web transfers while quickly delivering the requested content to end users. In previous work we proposed a new method for Web page retrievals that involves bundling of all embedded objects on a Web page into a single object [32]. The bundle method seeks to address the inefficiencies of HTTP/1.0 while avoiding the problem of inconsistent support for the features of HTTP/1.1. In this work we describe the approach and evaluate its use as a Web page retrieval method. We present the results of experiments performed using both simulated bundles from popular sites and a working bundle mechanism to compare bundling with existing retrieval methods.

The idea of bundles is intended to address problems with the current approaches to retrieving multiple objects needed to render a Web page. Versions 0.9 and 1.0 of the HTTP protocol are based on the model that uses a new TCP connection for each request/reply exchange with the Web server. To speed up the retrieval of content, many popular Web browsers open multiple concurrent TCP connections. Such a model, whether implemented with serialized or concurrent TCP connections, makes inefficient use of network and server resources. The overhead of setting up and tearing down each TCP connection contributes to router congestion. The operating system of the Web server also incurs per connection overhead and experiences TCP TIME_WAIT loading for each closed TCP connection. Response latency is affected because a new TCP connection is created for each request and each transfer independently goes through the TCP slow start phase.

Early on in the development of the Web this inefficiency of the simple HTTP model was realized and addressed by a number of proposals [24, 8, 18] advocating the use of a single TCP connection for multiple request/reply exchanges. Two new HTTP methods were also suggested, GETALL and GETLIST, which could be used to get all objects from a Web page in a single retrieval, and to get an arbitrary list of objects from a server in a single request respectively [21]. These efforts contributed to the decision to adopt persistent connections as default in the HTTP/1.1 protocol [7, 11].

HTTP/1.1 compliant clients and servers SHOULD (not MUST) (for the definition of these terms as applied to standards see [3]) permit persistent connections, although either side is allowed to terminate a persistent connection at any time. Both clients and servers can also indicate their unwillingness to hold a connection as persistent by including a "`Connection: close`" header as part of the HTTP request or reply respectively.

This flexibility in dealing with persistent connections was deemed necessary, particularly for servers, during the development of the HTTP/1.1 specification, but has led to mixed success in the current use of persistent connections. Studies have shown that 70-75% of Web servers at popular Web sites support HTTP/1.1 persistent connections, meaning that more than one object was successfully retrieved over the same TCP connection [10, 14].

However, support for persistent connections does not necessarily lead to reduced retrieval times for a set of objects from a server. One of the authors evaluated the impact of different strategies for using TCP connections to retrieve multiple objects from the same server on the end-to-end response time and found a number of interesting results [14]. First, persistent connections with serialized HTTP requests generally do not provide better response time than parallel requests. Second, persistent connections with pipelining generally provide better response than parallel requests, but pipelining was only supported by about 30% of the servers. Moreover, performance benefits are lost if TCP connections must be re-established.

Overall, persistent connections address inefficiencies associated with multiple concurrent connections, and when successfully used with pipelining can improve the response time. In practice, however, there is inconsistent support from Web servers, user agents, and intermediaries for persistent connections, particularly with pipelining. In this environment, browsers continue to open multiple concurrent TCP connections to the same server [28, 1], which leads to considering more efficient approaches for retrieving multiple objects from a server.

This work evaluates a new approach to the problem of delivering multiple Web objects from a single server to a client. Our approach is more efficient than using multiple concurrent TCP connections and more deterministic than a persistent connection carrying a variable number of client requests. The basic idea is for servers to group sets of related objects, such as objects needed to render a Web page, into a new object, called a *bundle* [32]. In its simplest form, a bundle is a concatenation of a URL string, HTTP headers, and content for each of its constituent objects. Servers advertise the availability of a bundle (or bundles) for each of their container (HTML) pages via a new HTTP response header `Bundles`. Clients understanding this header can then choose to issue a single HTTP request (over an existing or a new TCP connection) and fetch the entire bundle or to ignore the availability of the bundle and retrieve all remaining objects for the container page as is currently done. Upon obtaining the bundle, clients recover the objects encapsulated in it and cache each of them using the supplied URLs and HTTP headers. As clients render the page, the embedded objects should already be cached, but if a bundle does not contain all the necessary objects, then clients simply retrieve them from the server as is currently done.

An origin server or a server in a Content Distribution Network (CDN) can use this approach to serve multiple objects grouped together. In evaluating this approach, our focus has been on a set of objects necessary to fully render a Web page. In general, bundles could include objects that have some other form of relationship than being part of the same page. For example, all "popular" images at a site could be packaged into a bundle.

In the remainder of the paper we discuss the details of this approach and examine its impact on the entities involved in the handling of the HTTP requests and responses. We include results from a measurement study that examines performance impact if bundles were available from popular Web sites. We describe a server and client implementation of the bundle mechanism and present performance results under varying network conditions. We go on to examine the interaction between our approach and client side caching as well as content distribution. We conclude with a summary of our findings and a discussion of directions for future work.

## 2 Packaging Objects into Bundles

A common way to distribute software on the Internet is to package a collection of related files into a single file, using tools such as *tar* and *gzip*. We are also aware of Web sites that package a subset of their content (mostly static resources) using similar techniques and provide a pointer to the resulting file on their home pages (see [23] as an example). We are unaware, however, of any work that has proposed and evaluated a more selective and automated packaging of related objects.

In our approach the use of bundles is initiated by the content provider. In the simplest case, an origin server packages all objects embedded on a Web page into a bundle. Each object in a bundle is represented by its URL, relevant metadata (HTTP headers), and contents. HTTP headers associated with each object, *local headers*, are specific to that object. For example, an object can have `Content-Length` (used by the client in recovering the encapsulated objects) or `Last-Modified` HTTP header associated with it. When a bundle is requested by a client, the Web server also assigns it a set of HTTP headers, as it would for any other object. In the content of bundles, we will refer to such headers as *global headers*. In cases where global headers have the same names as the local headers, the local headers always take precedence over global headers. Otherwise, individual objects within a bundle inherit the global headers. Since all objects served by the same Web server naturally share some common HTTP headers, such as `Server` or `Date`, serving a single bundle instead of multiple individual objects results in a simple *header compaction* mechanism. With many small objects included in a bundle, byte savings could be significant.

For serving objects embedded in a Web page, a server can adopt a convention that bundles have the same names as their respective pages, except for the extension. For example, the bundle `http://www.wpi.edu/index.bndl` contains the embedded objects for the `http://www.wpi.edu/index.html` page. Bundles could be dynamically computed upon a client's request or could be precomputed. Dynamic construction of bundles is likely to incur substantial overhead due to a Web server parsing the page to determine which objects to include in the bundle. Precomputation avoids critical path processing but requires additional storage at the server since both bundled and original objects are stored. Popular objects, such as logos embedded on all site pages, will belong to many bundles, and should be separated into a separate bundle. Ultimately, servers decide if, when, and how to construct bundles.

One approach for constructing bundles is to group objects on a page, or across pages, based on their relationships and change characteristics, as discussed in [31]. Such grouping is reminiscent of the notion of volumes [16, 13, 5]. For example, static or infrequently changing objects on a page can be packaged together. Frequently changing objects on the same page can be organized into a separate bundle—or not grouped at all. While clients need to retrieve more than one bundle for a page in such cases, each bundle contains objects with similar change characteristics and cache control directives.

A sample interaction between a browser and a server using bundles is shown in Figure 1. When processing a request for a Web page that has an associated bundle, the Web server includes an additional HTTP header in its response, advertising the availability of bundles to the client. For example, the server includes the "`Bundles: index.bndl`" header when processing a request for `index.html`. If the requesting client chooses to take advantage of the available bundle, it issues a single HTTP request (over the existing or a new TCP connection) and indicates its ability to accept a new content type, `application/x-bndl`. Upon obtaining the bundle, the client extracts individual objects from it and reconstructs their metadata. These objects can then be stored in the local client cache (whether the client is a user agent or a caching proxy). At this point, the client can discard the bundle itself, although it may want to retain meta-information about the bundle for future retrievals (see Section 8). Clients that do not support bundles or choose to ignore them, fetch embedded objects as usual. The use of bundles is optional for both clients and servers, and it does not require any changes to the current method of content retrieval.

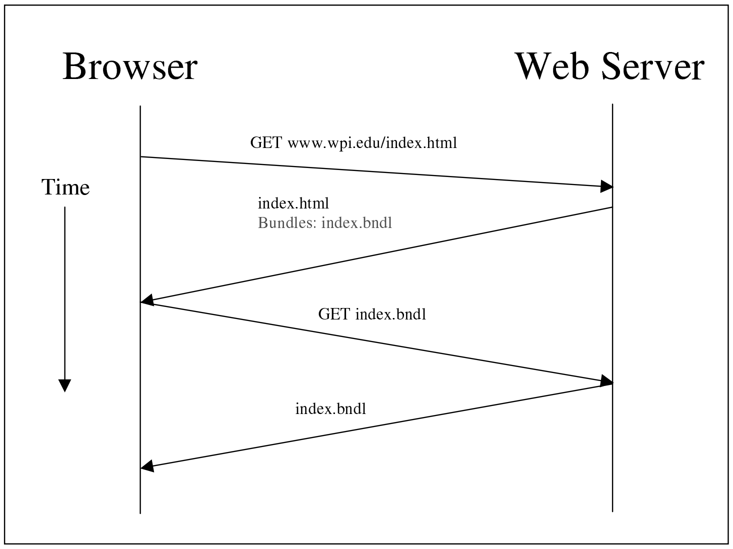

Figure 1: Basic HTTP Interactions with Bundles

An obvious extension of simply concatenating a set of objects together is to use a compression tool such as *gzip* to reduce the size of the bundle for transmission (potentially for storage as well). In the likely case that bundles primarily contain images, which are already compressed, opportunities for additional compression come mostly from textual HTTP headers within the bundle. In cases where bundles encapsulate textual objects, such as HTML frames, Cascading Style Sheets, or scripts to be executed on the client side (JavaScript that is not part of HTML, for example), applying compression could significantly reduce the size of the bundle.

## 3 Performance Impact on Servers

This section examines the performance implications that the use of bundles has on servers. The following section examines the performance impact for clients. Bundles are unlikely to find wide adoption unless shown to be beneficial to both clients and servers.

HTTP uses TCP as its transport protocol. As a connection-oriented protocol, TCP maintains state at each end point of the connection thus placing per-connection memory overhead on each peer host. With a large number of simultaneously active TCP connections, a Web server's memory requirements can grow large. A more significant per-connection overhead occurs when a host performs an active close on the TCP connection. After sending the final TCP ACK, that host must keep the closed connection in the TIME_WAIT state for twice the Maximum Segment Lifetime (MSL) [25]. MSL is specified as 2 minutes [22], but commonly used values are 30 seconds, 1 and 2 minutes [25]. Consequently, the local port number used by the connection that is currently in TIME_WAIT state cannot be reused for 1 to 4 minutes. In HTTP, Web servers are the ones normally closing the connection. Therefore, depending on the local port range made available by the operating system (it is not always 1024-65535), and the client request rate, servers might be unable to open new TCP connections, even if few are currently active. HTTP throughput reductions of up to 50% have been reported [6]. TIME_WAIT loading of busy Web servers is studied in [6].

Intuitively it is clear that Web servers should benefit from maintaining fewer TCP connections. And yet experience shows that support for persistent connections is not always implemented in Web server software or is deliberately turned off. The latter might be explained by the fact that lifetime of a persistent connection is *non-deterministic*—Web servers and clients are free to close a persistent connection when they deem necessary. The default connection-per-request model of HTTP/1.0, on the other hand, is *deterministic*—a server closes the connection after forwarding the response to the client. Our proposal utilizes a single TCP connection to effectively retrieve multiple objects while allowing the server to deterministically close the connection.

In addition to per-TCP connection overhead, Web servers also incur per HTTP request and per HTTP response overheads. Each HTTP request must be parsed, possibly logged, each HTTP response must be generated, including the HTTP response headers. Depending on the configuration and load, a Web server might need to fork a new process to service an incoming request.

Consider a Web server serving a Web page with *n* embedded objects under three different scenarios, as shown in Table 1. Under the connection-per-request model, used by HTTP/1.0 and HTTP/1.1 without persistent connection support, the number of TCP connections, HTTP requests, and HTTP responses directly depends on *n*. With the persistent connections, the number of HTTP requests and responses is still proportional to *n*. Only when bundles are used the number of connections and requests needed to retrieve a Web page is constant.

Table 1: TCP Connections and HTTP Requests/Responses Resulting From the Retrieval of a Web Page with $n$ Embedded Objects

| | Number of | |
|---|---|---|
| Scenario | Connections | Requests/Responses |
| Connection-per-Request | $n+1$ | $n+1$ |
| Persistent Connection | 1 | $n+1$ |
| Bundle Retrieved | 1-2 | 2 |

Additional overhead is incurred at the servers because bundles must be created for appropriate Web pages, which requires tracking changes to these Web pages. Storage to maintain bundles is also required. Web servers can control these costs by grouping objects into bundles according to object change characteristics. The set of rarely changing objects on a page would form one group. Frequently changing objects contained within a Web page can either be grouped in their own bundle or not grouped at all. In the latter case, clients retrieve these objects individually as is currently done.

## 4 Performance Impact on Clients

We use two methods to evaluate the effectiveness of the use of bundles for a client. One method is to implement the client- and server-side mechanisms required for the use of actual bundles, and to conduct experiments using these mechanisms in order to assess the practical implications of the use of bundles. This method is discussed in Sections 5-7.

Our other evaluation method involves measuring the time required to retrieve simulated bundles—existing objects that approximate as closely as possible the combined size of all of the objects embedded on a particular Web site's main page. This method allows us to measure and compare the retrieval times of Web pages from a large number of existing Web sites using both bundles and traditional retrieval mechanisms. An additional benefit of this method is that these measurements may be made without any client or server modifications.

For these experiments we constructed a list of 500 existing Web sites, including sites from the ".com", ".net", ".org", ".gov", and ".edu" domains. We used this list to perform a two-step process. The two steps were performed independently. In the first step, we used an automated tool to gather a list of existing objects of varying sizes at each Web site.

In the second step, we used other automated tools to retrieve the container page and embedded objects for each of the 500 Web sites. Each container page and associated embedded objects were retrieved using httperf [20] with four protocol options. Httperf is a publicly available tool that retrieves a set of objects from a server and reports statistics about the performance of the retrieval. Httperf provides the ability to retrieve objects using a variety of connection configuration options, including variations of the HTTP/1.0 and HTTP/1.1 protocols. To facilitate comparison of bundled retrieval times to retrieval times achieved by existing retrieval methods, in this study we used httperf with the following protocol options:

1. burst-1.0—retrieve objects using up to four parallel HTTP/1.0 requests,
2. serial-1.1—retrieve objects using serialized requests over an HTTP/1.1 persistent connection, and
3. burst-1.1—retrieve objects using pipelined requests over a single HTTP/1.1 persistent connection.

For each of the 500 Web sites, we also calculated the size of a hypothetical bundle for the page and retrieved (when possible) existing static objects with sizes that were the next smaller and the next larger from the set of objects found in the first step of our study.

The study with simulated bundles was done in April 2001 and October 2001 from a high-speed Internet client at Worcester Polytechnic Institute (WPI). The study was also conducted from a low-speed dialup client in October 2001.

Table 2 shows the averaged results for all of the sites for which we were able to identify both a smaller and a larger object to serve as simulated bundles, and thus were able to calculate expected bundle retrieval times based on interpolation between the retrieval times for the two objects. The Extrapolated Bundle column shows the expected bundle retrieval times calculated by extrapolating from the time to retrieve the smaller objects. The retrieval time values are in seconds. The value in parentheses following each retrieval time indicates the ratio of the retrieval time to the burst-1.0 retrieval time. The retrieval times shown for unbundled retrievals indicate the time required to retrieve all of the embedded objects, and do not include the time required to retrieve the container page.

Table 2: Overall Averages for Retrieval Time Compared to Burst-1.0

| | | | | | Ave. Retreival Time (Sec.) (Ratio with burst-1.0) | | |
|---|---|---|---|---|---|---|---|
| Test Run | Server Count | Ave. Size | Ave. # Objects | Ave. RTT | Burst-1.0 | Interp. Bundle | Extrap. Bundle |
| WPI Apr. '01 | 109 | 42226 | 17.6 | 0.222 | 2.17(1.0) | 0.95(0.4) | 1.05(0.5) |
| WPI Oct. '01 | 106 | 44653 | 18.7 | 0.100 | 1.78(1.0) | 0.81(0.5) | 0.78(0.4) |
| Dialup Oct. '01 | 94 | 40477 | 18.6 | 0.439 | 11.89(1.0) | 7.98(0.7) | 8.26(0.7) |

The results show that bundling embedded objects can provide significantly shorter elapsed embedded object retrieval times than the burst-1.0 protocol option. Due to bandwidth limitations, the reduction in retrieval times achieved using bundles is relatively less significant for dialup connections than for the faster connection type.

Table 3 compares the retrieval time for simulated bundles to the retrieval times using the burst-1.1 protocol option. The retrieval times for other protocol options are shown for comparison. This table includes the averaged results for all of the sites for which we were able to successfully retrieve all of the embedded objects in a single TCP connection using the burst-1.1 protocol option. In order to increase the number of samples shown, the table excludes results from sites for which we were unable to identify a larger object to serve as a simulated bundle. Consequently, Table 3 does not show interpolated results. The extrapolated bundle retrieval column shows the expected bundle retrieval time calculated by extrapolating from the time to retrieve the smaller object. The value in parentheses following each retrieval time indicates the ratio of the retrieval time to the burst-1.1 retrieval time.

Table 3: Overall Averages for Retrieval Time Compared to Burst-1.1

| Test Run | Server Count | Ave. Size | Ave. # Objects | Ave. RTT | Ave. Retrieval Time (Sec.) (Ratio with Burst-1.1) Burst-1.0 | Serial-1.1 | Burst-1.1 | Extrap. Bundle |
|---|---|---|---|---|---|---|---|---|
| WPI Apr. '01 | 100 | 53646 | 17.0 | 0.169 | 1.87 (1.3) | 2.34(1.7) | 1.40(1.0) | 1.24(0.7) |
| WPI Oct. '01 | 110 | 55924 | 17.8 | 0.087 | 1.80 (1.2) | 1.96(1.3) | 1.55(1.0) | 0.99(0.5) |
| Dialup Oct. '01 | 131 | 51452 | 15.0 | 0.506 | 13.65 (1.0) | 15.23(1.1) | 13.46(1.0) | 11.52(0.8) |

The improvement for bundles relative to the burst-1.1 option is not as significant as for burst-1.0. This result is expected because if the server is able to continue to send content in response to pipelined requests with the burst-1.1 option then performance should be comparable to that of bundles. Although not shown, we did break down the results in Tables 2 and 3 according to the number of embedded objects, number of embedded bytes, and round trip time (RTT). We found that bundling has more relative effect for pages with more objects. We found it has less relative effect based on differences in the number of bytes or RTT [27]. The overall results from this study confirm results found in a preliminary study based on older measurements [32].

## 5 Implementation

Our implementation consists of two parts: a bundle-capable Web server, and a thin client-side Web proxy that is used in conjunction with a Web browser such as Netscape Navigator or Internet Explorer. This architecture is illustrated in Figure 2. The Web proxy and the Web browser are co-resident on the client machine and work in tandem to handle the retrievals of individual and bundled objects. The Web server is located on a server machine and serves both bundled and unbundled pages. The remainder of this section details the implementation.

### 5.1 Bundle Construction

In our implementation, a bundle is a file containing object-specific headers, a URL string for the object, and object contents for each of the bundled objects. Figure 3 shows an example of a

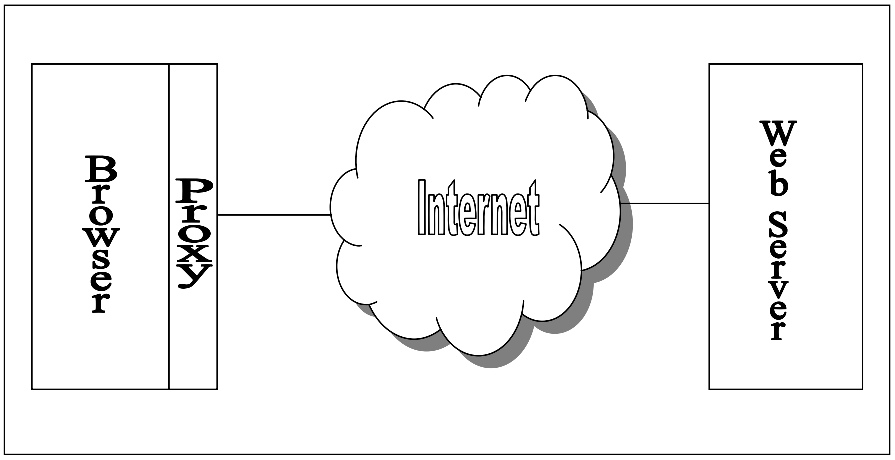

Figure 2: Implementation Architecture

bundle. For each object in the bundle, we first list any headers that are specific to the object, such as `Content-Type`, `Content-Length`, and `Last-Modified`.

Following the object-specific headers are a URL string for the object and a line containing only a carriage-return line-feed character sequence. The URL string and the carriage-return line-feed character sequence are used by the proxy during the bundle extraction process. The objects contents are copied into the bundle file starting on the line immediately following the carriage-return line-feed sequence. Data for any additional objects immediately follows.

Our implementation is capable of handling compressed bundles. A compressed bundle is formed by creating a bundle using the above-mentioned technique, then compressing the resulting bundle file using the *gzip* compression utility. The Web server handles compressed bundles in exactly the same manner as uncompressed bundles. The proxy uses zlib compression library calls to detect compressed bundles and uncompress them on the fly.

## 5.2 Bundle-Aware Web Server

The Web server in our implementation is a simple server written in Java. Because we wanted to eliminate the effect of server-side caching during our retrieval measurements, our Web server does not perform any caching of recently served objects.

```
Content-Type: image/gif
Content-Length: 866
Last-Modified: Tue, 27 Nov 2001 03:05:46 GMT
Url: http://grover.wpi.edu:8080/HTML/Buttons/about-on.gif
\r\n
<<Object contents>>
          .
          .
          .
Content-Type: image/gif
Content-Length: 2878
Last-Modified: Fri, 26 Oct 2001 04:13:42 GMT
Url: http://grover.wpi.edu:8080/images/parchment.gif
\r\n
<<Object contents>>
          .
          .
          .
```

Figure 3: Bundle Construction

The Web server accepts a command-line argument that indicates the name of a ”bundle map” text file. Each line is this file contains two items: the fully qualified pathname of a file or object that may be requested by a client, and the URL of a corresponding bundle object. The Web server reads this file at startup, and stores its contents in a structure in memory. Upon receiving a request from a client, the Web server first checks the bundle map to determine whether the request matches any of the entries in the bundle map. If a match is found, the Web server adds the `Bundles` header to the response, thus notifying the client of the availability of a bundle object that is associated with the requested page. The Bundles header includes the URL of the bundle object associated with the request. If the Web server receives a request for a bundle object, the object is returned to the client exactly as any other object would be.

## 5.3 Client-Side Web Proxy

Ideally, the client side implementation of the bundle mechanism would consist of an industry-standard Web browser modified to handle bundles. For this study, we chose to implement our client-side mechanism in a way which would approximate as closely as possible the use of a standard browser while keeping the implementation manageable.

Our implementation approximates a modified standard Web browser by placing a small Web proxy in front of an unmodified Web browser. The browser and proxy processes run simultaneously on the client machine. All incoming and outgoing Web traffic passes through the proxy, enabling the proxy to intercept bundle-related traffic and perform bundle-handling operations on behalf of the browser. The combined browser and thin proxy provide a reasonable approximation of a browser modified to handle bundles.

Our client-side Web proxy is written in Perl. The proxy communicates with the Web browser via a TCP connection. The proxy implementation includes a file system-based cache in which recently retrieved objects may be stored. The proxy is designed to handle both uncompressed and compressed bundles. Compressed bundles are uncompressed on the fly as the bundle stream is received via the TCP connection to the server. Similar to proxies such as Squid [29], our proxy supports only non-persistent connections for straight-forward and less error-prone implementation.

Figure 4 illustrates the interaction between the Web browser, proxy, and Web server during retrieval of a container page and associated bundle. The process begins when the browser issues a request for the container page. The request passes through the proxy and is forwarded on to the Web server. When the Web server receives the request, it checks the bundle map to see if a bundle exists for this request. If a match is found, the Web server advertises the availability of a bundle by attaching the `Bundles` header to the container page as it is returned to the client.

The container page passes first through the proxy on the way back to the Web browser. As it forwards the container page response on to the browser, the proxy scans the headers on the container page to check for the presence of the `Bundles` header. If the `Bundles` header is present, the proxy issues a request to the Web server for the bundle object indicated in the `Bundles` header.

The Web server responds to the proxy's request for the bundle object by returning the bundle object as it would any other object. As the proxy receives the bundle object back from the Web server, the proxy extracts the individual objects from the bundle and places them into the proxy cache. If the bundle is in a compressed form, the proxy uncompresses the bundle as it is being received via the TCP connection with the server.

As each object is read from the bundle during the extraction process, the proxy stores the object in the proxy cache using the provided URL string as a key. When the proxy subsequently receives a request from the Web browser for an object, the proxy first queries the cache using the URL requested by the browser. If a key matching the requested URL is found, the associated object is

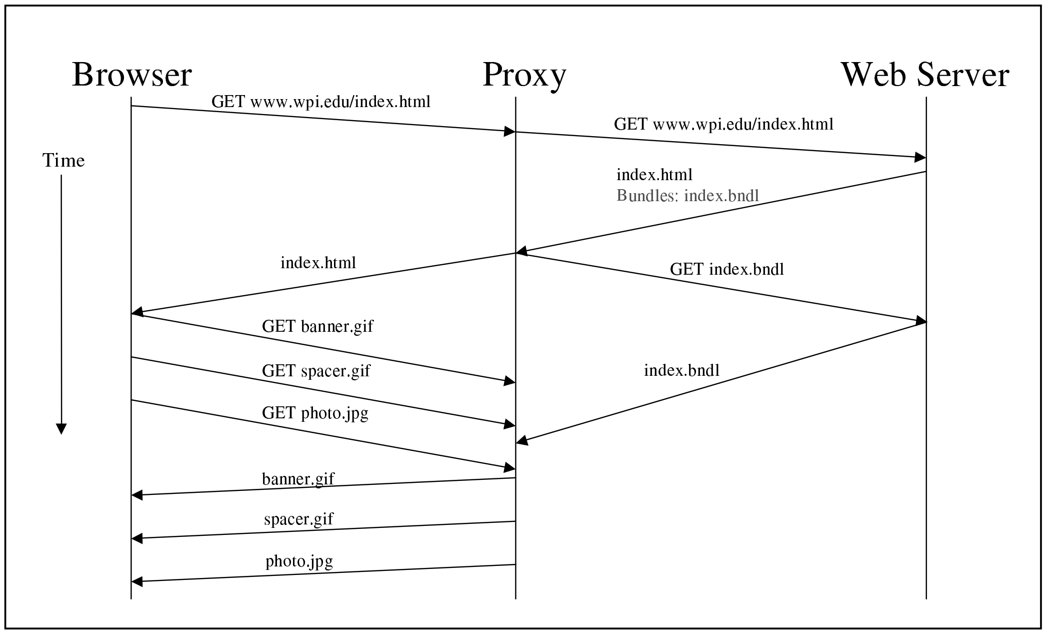

Figure 4: Implementation Transaction Diagram

retrieved from the proxy cache instead of being retrieved from the server.

The contents of an object within a bundle starts after a line containing only the carriage-return line-feed character sequence. The corresponding end of the object's contents is identified using the value of the `Content-Length` header.

The proxy returns the container page to the browser and immediately issues the request for the bundle object. Consequently, while the proxy is waiting for and receiving the bundle object from the Web server, the browser receives the container page, parses the container page, and issues requests for any embedded objects.

During this period bundle processing occurs in the proxy. The proxy is receiving and processing a bundle that may contain the embedded objects for the associated container page. Therefore, the proxy must carefully handle any requests received from the Web browser while the bundle extraction process is proceeding.

If the proxy receives a request for an object during the bundle reception and extraction process, the proxy first checks to see whether the requested object is in the proxy cache, since the requested object may have already been extracted from the incoming bundle. If the requested object is in the cache, the object is returned immediately to the browser. If the object is not in the proxy cache, the proxy does not immediately forward the request on to the server. Instead, the proxy adds the request to a list of outstanding requests. After each object is extracted from the bundle, the proxy checks to see whether any outstanding requests exist for the just extracted object. If there are any outstanding requests for the object, the proxy immediately returns the object to the Web browser. After the proxy has finished processing the bundle, any requests still outstanding are then forwarded on to the appropriate server(s). This logic ensures that requests for embedded objects that are contained in the bundle are fulfilled by the bundled objects, and requests for embedded objects that are not contained in the bundle are also satisfied.

## 6 Implementation Experiments

We performed experiments using our bundle implementation in order to assess the impact of bundle use on Web page retrieval performance. In order to compare the performance of a working bundle implementation to existing retrieval techniques, we retrieved pages containing a varying numbers of embedded objects of varying sizes using three different network connection types. The embedded objects were retrieved in both unbundled and bundled form. Retrieval times were measured and recorded.

Our test sample consisted of three Web pages representing a small, medium and large Web page in terms of number and collective size of embedded objects. The pages were selected from existing pages on the WPI site. The container pages and associated embedded objects were copied to our test Web server. The characteristics of the three pages and the bundles formed by their embedded objects were as follows:

1. Small page: 10 embedded objects, 26.4k bytes uncompressed bundle size, and 23.4k bytes compressed bundle size.

2. Medium page: 28 embedded objects, 69.1k bytes uncompressed bundle size, and 59.9k bytes compressed bundle size.

3. Large page: 47 embedded objects, 113.6k bytes uncompressed bundle size, and 99.0k bytes compressed bundle size.

The embedded objects on these pages included images, style sheets, and JavaScript. However, since most of the embedded objects were images little reduction in size was achieved by compressing the bundles composed of these objects. The specific object counts and sizes that we chose correlate well with the findings of a study done in April, 2002 by Krishnamurthy et al., in which they divided the number of embedded objects and bytes for frequently requested pages into 33% and 67% percentiles, producing three ranges for each of the two metrics [12]. The three ranges for embedded object counts were less than 7, between 8 and 22, and greater than 22. The three ranges for embedded object bytes were less than 20K, between 20K and 55K, and more than 55K. The small, medium, and large pages that we chose respectively fall into the lower portion of the middle percentiles, the lower portion of the upper percentiles, and well into the upper percentiles in the data gathered by Krishnamurthy et al.

The Web server was located at WPI for all tests. The same client machine was used for all tests, but the client machine was moved to three different geographical locations in order to perform tests using dialup, cable modem, and T1 connection types.

For each of the three client connection types, each of the three pages and associated embedded objects were retrieved using the following methods:

- unbundled, using up to four concurrent HTTP/1.0 connections,
- uncompressed bundle, and
- compressed bundle.

Ten retrievals were performed for each page for each method using each of the network connection types. The retrievals were performed by a set of automated test tools developed for this experiment. All of the retrievals for each connection type were performed during a single continuous run in order to minimize the effect of network or server load on the results. For the same reason, within each test run, retrievals were performed using the three retrieval methods in cyclical order. The Web server was lightly loaded during testing.

The tests used Netscape Communicator as the Web browser in conjunction with the thin Web proxy and Web server developed for our bundle implementation. We used a script to launch Netscape and initiate each retrieval. Netscape was stopped and restarted between each retrieval to eliminate the effect of browser caching and ensure consistency of the environment between retrievals.

# 7 Implementation Experiment Results

In analyzing the results of experiments performed using our bundle implementation, we examined the effect of the use of compressed and uncompressed bundles on the retrieval characteristics: elapsed time for retrieval of the container page and all embedded objects, time of availability of individual embedded objects during Web page retrieval, and average delay per embedded object. We present the results of our experiments and analysis in this section.

## 7.1 Elapsed Time

Figures 5, 6, and 7 show the average elapsed retrieval times of the three sample pages using the dialup connection, cable modem connection, and T1 connection, respectively. The elapsed times indicate the time required to retrieve the container page and all of the embedded objects. The results in these figures are grouped by sample Web page. Each figure shows the average elapsed time to retrieve each of the three sample Web pages using each of the three configurations - unbundled, bundled/uncompressed, and bundled/compressed. The error bars in each figure show the 95% confidence interval above and below the average value.

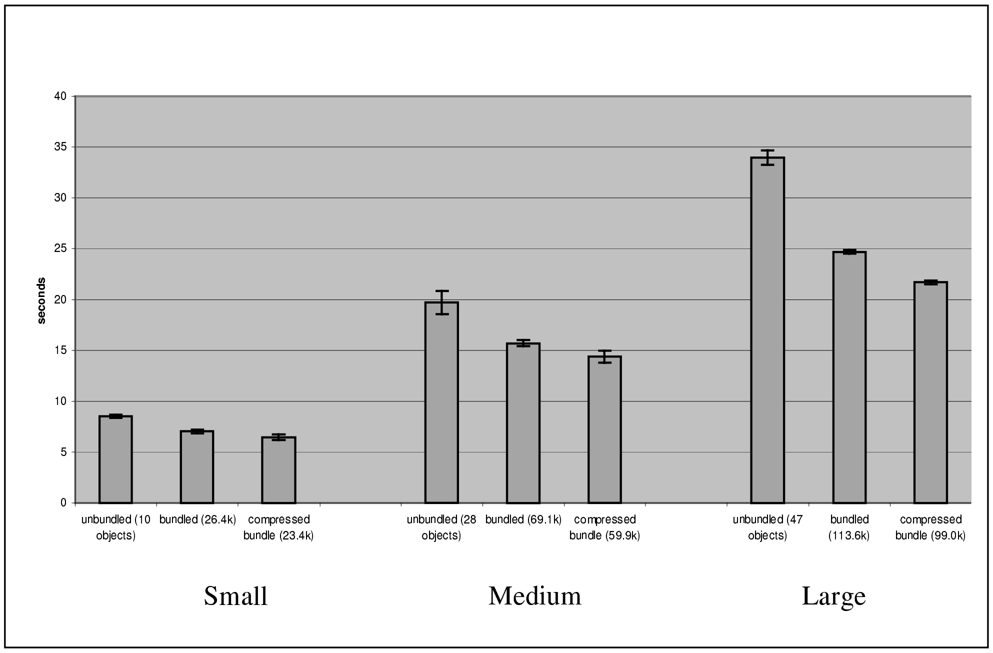

Figure 5: Average Elapsed Time for Page—Dialup Connection

As Figure 5 shows, the average elapsed times for both uncompressed and compressed bundles were significantly shorter than the average retrieval times for unbundled retrievals using multiple concurrent HTTP/1.0 connections over a dialup connection. Since the compressed bundle sizes were similar to the uncompressed bundle sizes, the average elapsed times for compressed bundle retrievals were not significantly shorter than for uncompressed bundle retrievals.

Figures 6 and 7 show the average elapsed times for the cable modem and T1 connection tests, respectively. For both connection types, uncompressed and compressed bundle retrieval times were significantly shorter than unbundled retrieval times.

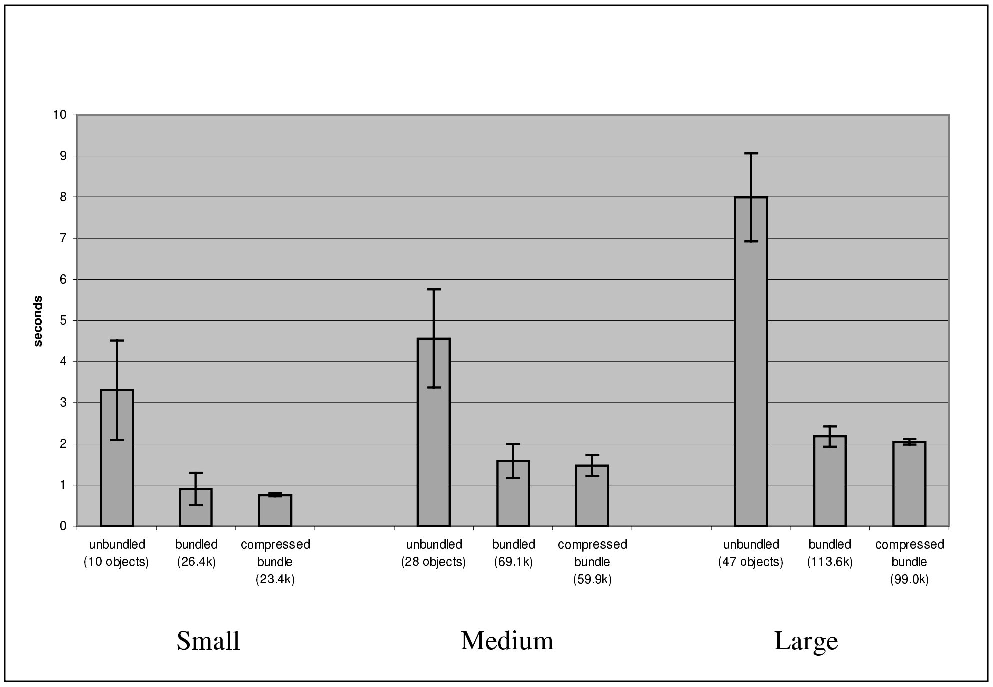

Figure 6: Average Elapsed Time for Page—Cable Modem Connection

In Figure 7 the average elapsed time and confidence interval for the unbundled retrieval of the Large page are influenced by a single outlying retrieval time value of 16.3 seconds. If this outlier were excluded, the average elapsed time would be 6.0 seconds with a 95% confidence interval of 0.2 seconds.

Figures 5, 6, and 7 show that bundled retrievals achieve more significant relative reductions in retrieval times for faster connection types than for dialup connections. As previously mentioned, this difference may be attributed to the fact that bandwidth limitations are a greater factor in retrieval time performance for dialup connections as compared to faster connection types.

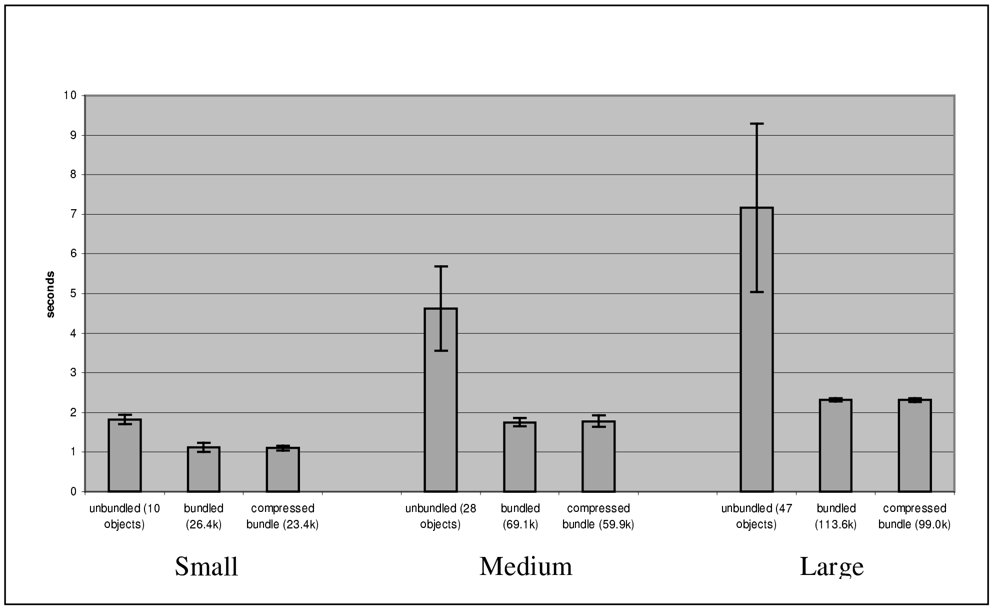

Figure 7: Average Elapsed Time for Page—T1 Connection

## 7.2 Time of Availability of Individual Objects

In addition to comparing the overall elapsed times of bundled and unbundled retrievals, we also examined the effect of bundle use on time of availability of individual embedded objects during retrieval of a Web page. During retrievals using HTTP/1.0 and HTTP/1.1, embedded objects are requested individually and become available to the Web browser throughout the retrieval time period. Although the retrieval times are affected by the relative sizes of the embedded objects and network and server load, in most cases we would expect the times of availability to be distributed fairly evenly throughout the retrieval period when using existing retrieval methods.

A naive bundle implementation might provide for a shorter overall retrieval time while clustering the times of availability of the embedded objects at the end of the retrieval period. From the user's perspective, a retrieval in which the objects on the page appear in an evenly distributed manner throughout the retrieval period might be preferable to having to wait until the end of the retrieval period for any of the objects to be available, even if the latter case produced a shorter overall retrieval time.

The first two sets of points in Figure 8 show the time of availability of individual embedded objects for a typical unbundled retrieval of our small sample Web page using a dialup connection. The data in the chart were recorded using our implementation components. For each embedded object, the figure shows the time when the request for the object was received at the proxy, and the time when the proxy had finished returning the object to the browser.

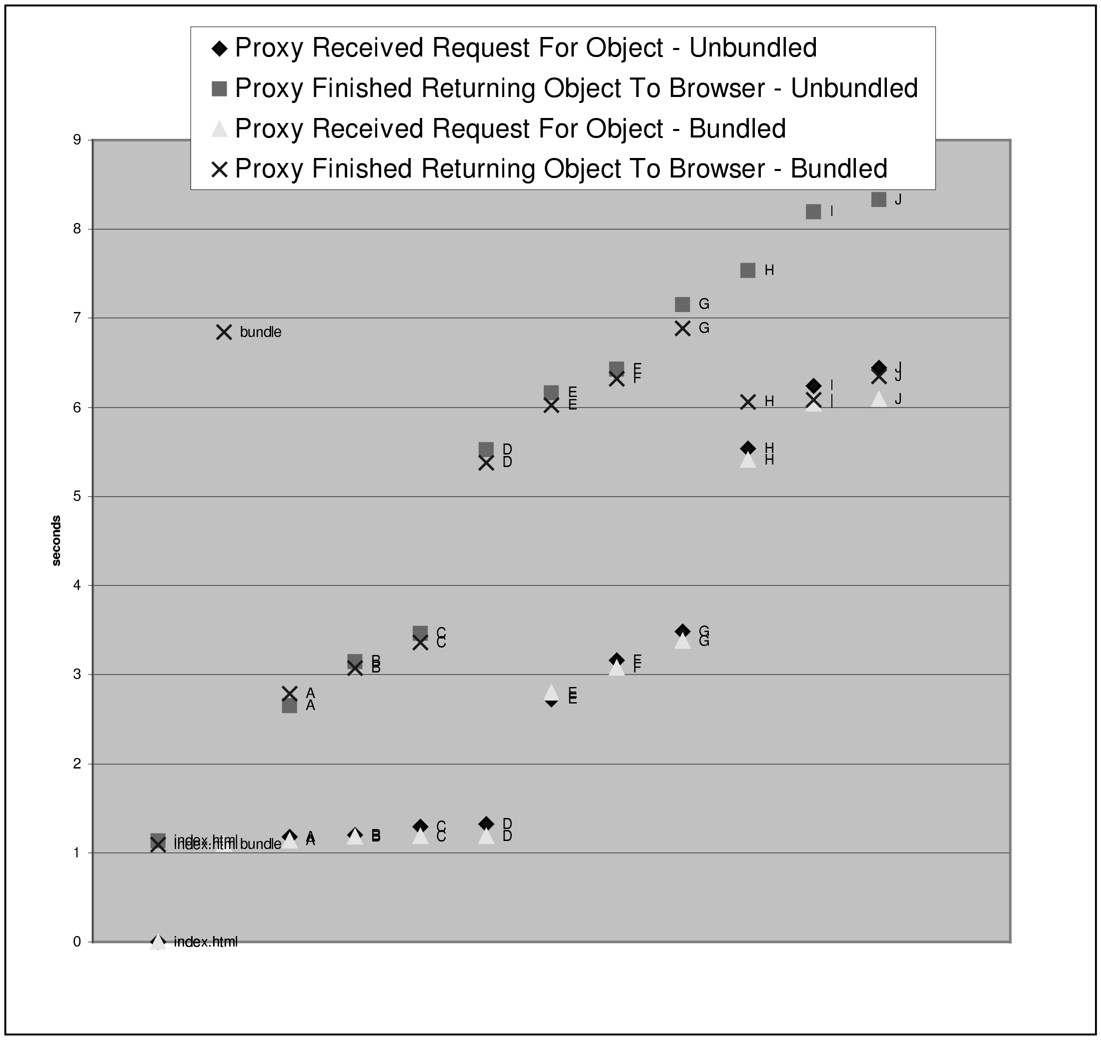

Figure 8: Object Availability Comparison, Dialup Connection, Small Page

After retrieving the container page, the browser issues four concurrent requests for the first four embedded objects, indicated in the figure as objects ’A’, ’B’, ’C’, and ’D’. After each object is received, the browser issues a requests for another object, thereby maintaining four outstanding concurrent requests at all times. This figure illustrates the relatively even distribution of times of availability of individual embedded objects throughout the retrieval period using traditional retrieval techniques.

The other two set of points in Figure 8 show the time of availability of individual embedded objects for a typical bundled/uncompressed retrieval of the same small sample Web page using a dialup connection. Immediately after retrieving the container page, the proxy begins retrieving the bundle on behalf of the browser (bundle request “triangle” is obscured in the figure). The figure also shows that our bundle implementation allows individual objects to be extracted from the bundle and returned to the browser while the bundle stream is being received from the Web server. By using this capability we avoid the problem of delaying the availability of embedded objects until the end of the bundle retrieval period.

As the figure shows, the browser requested most of the objects before the proxy had extracted them from the bundle. The extremely short timing interval for object ’I’ shows an example of the case of an object being extracted from the bundle and placed in the proxy cache before being requested by the Web browser. When a request for the object is subsequently received at the proxy, the object is immediately returned to the browser.

Comparing the unbundled versus bundled results in Figure 8 shows the time of availability for objects ’A’ through ’G’ is nearly identical for the bundled and unbundled retrievals. For objects ’H’ through ’J’, which were requested by the browser near the end of the bundle retrieval period, the time of availability is earlier for the bundled retrieval than for the unbundled retrieval. These results indicate that the time of availability of individual objects in bundled retrievals is as good as or better than that of unbundled retrievals.

## 7.3 Average Delay Per Embedded Object

We also compared the average delay per embedded object for bundled and unbundled retrievals as measured during our elapsed time tests. For each embedded object, the delay was calculated as the difference between the time when the object was available to the browser and the time when the last byte of the container page was received. The time measurements were made at the proxy in our implementation architecture. For each Web page retrieval, the average delay per object was calculated by averaging the delay values for all of the embedded objects in the page.

Figures 9, 10, and 11 show the average delays per embedded object in our tests using a dialup connection, cable modem connection, and T1 connection, respectively. Each bar in these figures represents the average of the average delay per object calculated for the 10 retrievals performed for that particular configuration. The error bars in each figure show the 95% confidence interval above and below the average value.

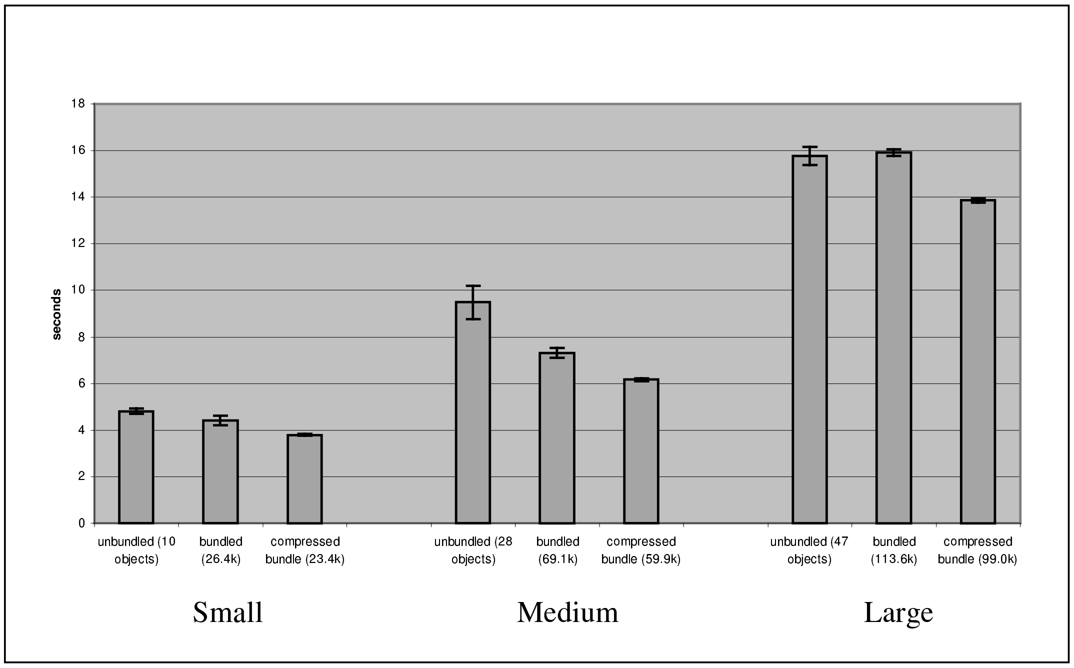

Figure 9: Average Delay Per Embedded Object—Dialup Connection

As Figure 9 shows, the average delay per embedded object for both compressed and uncompressed bundled retrievals was the same as or shorter than that measured for unbundled retrievals using a dialup connection. Figures 10 and 11 show the results for cable modem connection and T1 connection, respectively. These figures show that for cable modem and T1 connections, the average delay per embedded object was significantly shorter for the bundled retrievals than for the unbundled retrieval for every page type.

In Figure 11 the average delay and confidence interval for the unbundled retrieval of the ”Large” page are influenced by a single outlying delay value of 12.8 seconds. If this outlier were excluded, the average delay would be 2.9 seconds with a 95% confidence interval of 0.1 seconds.

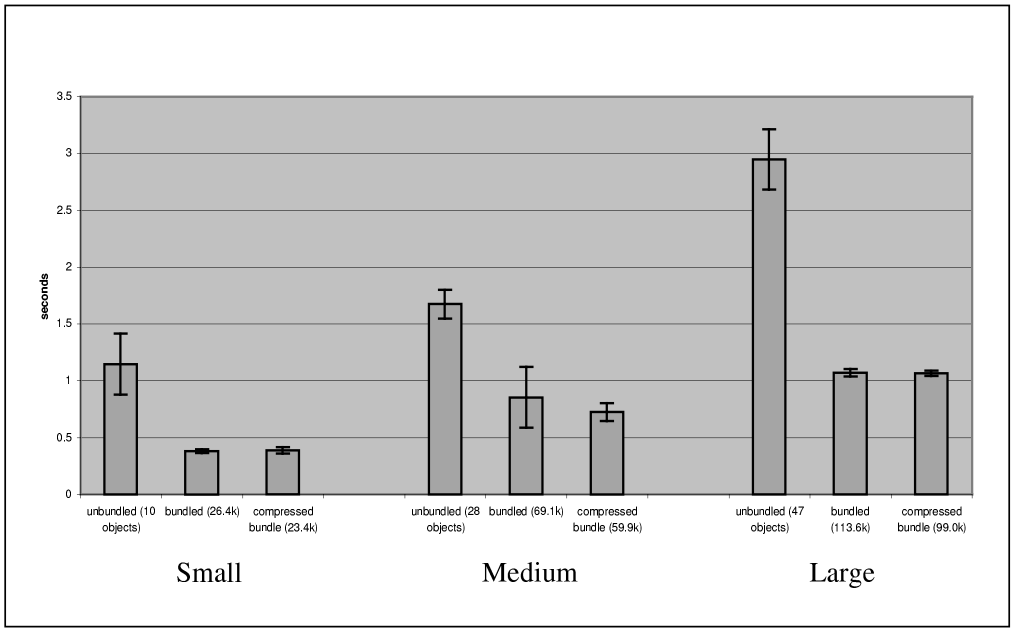

Figure 10: Average Delay Per Embedded Object—Cable Modem Connection

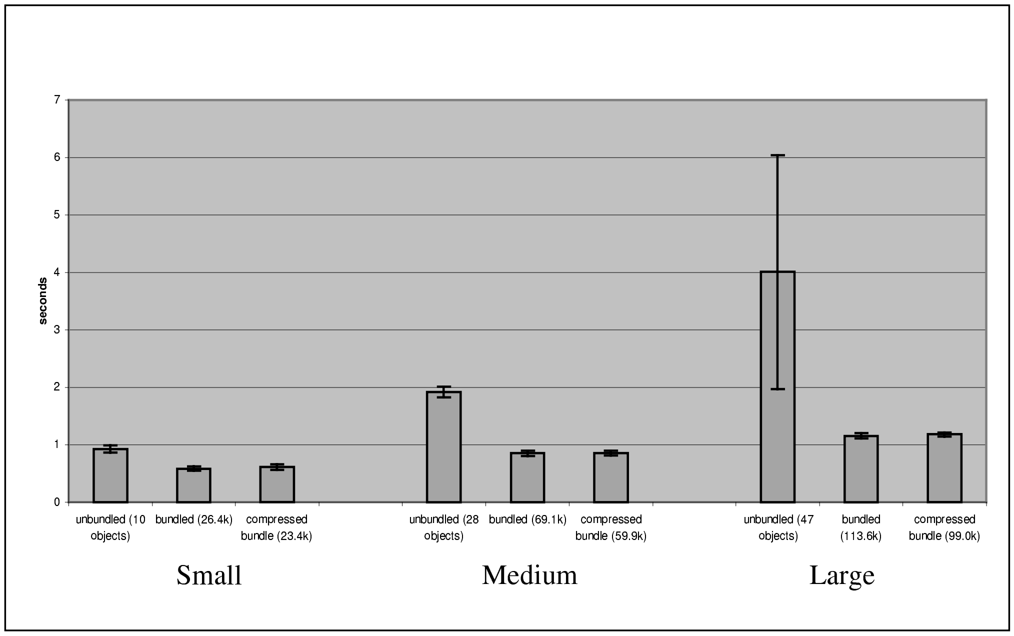

Figure 11: Average Delay Per Embedded Object—T1 Connection

### 7.4 Implementation Results Summary

In summary, tests performed using our bundle implementation indicate that use of bundles significantly reduces the elapsed time required to retrieve the container page and embedded objects for pages of varying sizes and containing varying numbers of objects using dialup, cable modem, and T1 connections. In addition, a properly designed bundle implementation appears to result in times of availability of individual embedded objects that are equivalent to or better than those achieved with traditional unbundled retrievals. The average delays per embedded object achieved when using both compressed and uncompressed bundles are equivalent to or shorter than the average delays incurred by unbundled retrievals.

## 8 Interaction with Caching and Content Distribution

The interaction of bundling with caching and content distribution are two issues that we did not specifically address in our implementation, but have examined in some depth. A passive Web cache, whether it is a browser cache or a proxy cache, can only serve objects which have been previously retrieved, subject to per-object restrictions. The first retrievals of a Web page and its associated objects would result in cache misses and hence the availability of a bundle containing all embedded objects for the page would be of value to the client as shown in our results. As the client subsequently retrieves the same or other bundles from the server, they might contain objects that the client already has in its cache. We see three approaches addressing this negative interaction between bundles and caching: validating individual objects, validating entire bundles, and delta encoding of bundles.

### 8.1 Validating Individual Objects within a Bundle

The HTTP response header, rather than simply advertising the bundle, as "`Bundles: index.bndl`," could also include a list of objects encapsulated in the bundle and their sizes, as

```
Bundles: index.bndl (main.css size=sz1, img1.gif size=sz2,
img2.gif size=sz3);
```

This list allows a client cache to decide whether the new bundle contains enough new objects to warrant its retrieval. If not, the client should ignore the bundle and retrieve individual objects as needed. Servers could also include other validators, such as `Last-Modified` or `ETag`, to aid clients in determining if new object retrievals are needed.

## 8.2 Validating Entire Bundles

In the second approach, a client cache retrieves a bundle object, extracts and stores its encapsulated objects, discards the bundle itself, but retains its HTTP headers, especially the ones used for cache validation. For example, if the `Last-Modified` or `ETag` headers are present, the cache could subsequently use them in a `GET If-Modified-Since` (IMS) or a `GET If-None-Match` request to the server to verify the freshness of the bundle. If the bundle has not changed, the server will indicate so with the HTTP response status code `304 Not Modified`. In this case, the client cache would have automatically validated multiple objects by issuing a single *aggregate* IMS request. If the bundle has changed, the server will reply with the HTTP status code `200 OK` followed by the body of the new bundle. If the bundle has changed due to only a subset of its encapsulated objects changing, its retrieval wastes resources. This problem leads to the third approach.

## 8.3 Delta Encoding of Bundles

The third approach uses delta encoding. The idea of the technique, first published by Williams et al. [30], is for the origin server (or proxy) to compute a difference (delta) between an old and a new copy of an object and communicate that difference to the client, provided that the client has that old copy of the object. The client can construct the new object by applying the delta to the old object. Mogul et al. [19] quantified the benefits of end-to-end delta encoding and compression and proposed extensions to the HTTP protocol to support the technique. These extensions are detailed in a published Internet Draft [17].

Since bundles are regular objects, the delta encoding technique can be uniformly applied to bundles just like it can be applied to any other object. Example delta encodings include those produced by UNIX *diff -e* or by *vcdiff* [9]. Delta encoding of bundles could also be implemented as a modified *rsync* mechanism [26]. This mechanism efficiently computes differences between two instances of the same file and is commonly used to update software packages produced with tools such as *tar*.

Upon receiving the delta encoded response, the client first reconstructs the older version of the bundle, based on the preserved meta data and individual cached objects, and then applies the delta to the result to produce the new version of the bundle. Problems arise when one or more of the objects from the old bundle are evicted from the client's cache, making the reconstruction of the old version of the bundle impossible. To address this problem, clients could cache bundle contents instead of discarding them or use a bundle-aware differencing algorithm.

The output of a bundle-aware differencing algorithm, given a new and an old versions of a bundle, encapsulates all objects in the new version that are not part of the old version. It also includes all modified objects in the new version. In addition, the output includes a list of objects that were part of the old version, but are not in the new version, to help clients differentiate between removed and unchanged objects. Given that bundles are likely to contain mostly images, computing differences on an object-by-object basis makes sense. When the client receives such a bundle-aware delta, all it needs to have in its cache is the unchanged objects from the old bundle. If some of them were evicted from the cache, the client can retrieve them individually.

We now illustrate how bundle-aware deltas can be specified within HTTP. Suppose a client obtained a bundle `index.bndl` from `www.wpi.edu`, cached all the objects encapsulated in it, discarded the bundle itself, and kept meta data for the bundle. If at a later time this client wants to obtain the current value of the bundle it can send the following request to the server (this example is adopted from [17]):

```
GET /index.bndl HTTP/1.1
Host: www.wpi.edu
If-None-Match: Bundle-ETag1
A-IM: diffe, vcdiff, bsync, gzip
```

In this example, the client indicates that it has a cached copy of the bundle, identified by `ETag` with the value `Bundle-ETag1`. Delta encoding is considered to be an instance manipulation, and the `A-IM` HTTP request header, short for `Accept-Instance-Manipulation`, indicates which delta encodings the client supports. The client can accept delta updates produced by UNIX *diff -e* and *vcdiff*. The *bsync* specification, short for “bundle sync,” represents the previously described bundle-aware encoding algorithm. To our knowledge, there is no existing tool that does the proposed bundle differencing, although similar distribution synchronization tools exist. The client also indicates that it can accept compressed responses using *gzip*, whether or not they were delta-encoded.

If the entity tag for `index.bndl` has changed (say, to `Bundle-ETag2`), the server, if it supports delta encoding, will compute the difference between the current version of the bundle and the older one, whose `ETag` value is `Bundle-ETag1`, and send the response to the client, as shown below:

```
HTTP/1.1 226 IM Used
Server: Delta-Aware Server/1.0
ETag: Bundle-ETag2
IM: bsync
Date: Wed, 29 May 2002 10:00:00 GMT

......body of the new bundle......
```

A new HTTP response code `226 IM Used` is required to force HTTP/1.0 proxies to forward all instance-manipulated responses without storing them (other options are discussed in [17]). If the server does not support delta encoding or does not implement any of the algorithms supported by the client then it replies with the HTTP response code `200 OK` and the body of the entire new bundle.

## 8.4 Interaction with Intermediate Caches

We also considered the issue of bundle content handling by intermediate caches between a client and server. A common example would be a proxy cache through which all client requests are directed. In the case that the proxy cache is not aware of bundles, but the client is, then the bundle object is simply treated as an object that the proxy may or may not cache according to its own policies. In the case that the proxy cache is aware of bundles, then the proxy could interrogate server replies and retrieve bundle objects in parallel with sending responses back to the client. This approach is similar to that taken by the proxy cache in our implementation.

## 8.5 Interaction with Content Distribution

The other potential negative interaction is the use of bundles with the distribution of Web content across multiple servers. Results obtained in November, 1999 by Krishnamurthy and Wills showed a relatively small fraction (15%) of 700 popular Web sites using more than one server to serve content for their home pages [14]. However, in April, 2002 we did a follow-up study of the home pages for 147 popular sites [15] (not a strict subset of the previous set) and found that 86% of these sites used more than one server to serve the objects on the home page. We address the apparent disconnect between this trend to distribute content and our proposal of consolidating it in two ways.

First, the distribution of relatively small number of objects to different servers does not appear to be a good idea in terms of performance, regardless of whether bundles are available or not. Each new server requires a new DNS lookup on the part of the client as well as opening a new TCP connection.

Second, if many objects are served by auxiliary servers, for instance by a special "image server" at a site or by a CDN server, then these objects are good candidates to be packaged together into a single object. A bundle can be retrieved from a CDN server just as it can be from an origin server. A bundle could also contain objects assigned to multiple servers if these objects were all under the control of a single content provider.

## 9 Related Work

We are aware of one existing commercial product which is similar in some regards to the bundle concept. Like the bundle concept, the wwWhoosh WebFlight Content Accelerator developed by wwWhoosh, Inc. aims to reduce the number of TCP connections and amount of data transferred across the Internet during Web page downloads by repackaging Web page content and transmitting multiple objects in a single response [33].

The wwWhoosh product differs from the bundle concept in two key ways. First, their product uses a separate proprietary server in between the origin server and the client. This proprietary server retrieves objects from the origin server, repackages and compresses the objects, and serves the repackaged data to wwWhoosh clients. In contrast, the bundle concept proposes serving bundles directly from the origin server and transmitting bundles as standard HTTP response objects.

The second and perhaps more important difference between the wwWhoosh product and the bundle concept is the way in which the availability of a multi-object unit is indicated to the client. In the bundle concept, a server advertises the availability of a bundle via an HTTP header. The wwWhoosh product uses a dynamically updated list of available repackaged objects. This list must be retrieved by the client from the proprietary intermediate server as part of the Web page download process. The list is retrieved in a separate transaction, thereby necessitating an additional request/response exchange. Once retrieved, the list is used to index into a database of repackaged objects stored on the intermediate server.

## 10 Summary and Future Work

In this paper we have examined a new approach to retrieving multiple Web objects. In our approach, servers package related objects at a site into a bundle object, which can be obtained by clients using a single HTTP request/response exchange. This approach lessens the need for clients to use parallel requests or try and maintain persistent connections with the server because fewer objects need to be retrieved.

We have found that the use of bundles for Web page retrievals generally results in shorter overall elapsed embedded object retrieval times as compared to both the burst-1.0 and burst-1.1 protocol options. Our implementation experiments have shown that a working bundle mechanism can provide significantly shorted elapsed retrieval times than burst-1.0 retrievals for pages containing varying sizes and numbers of embedded objects. Shorter elapsed retrieval times have been shown to be achievable using dialup, cable modem, and T1 client-side connections. Since many of the embedded objects in typical Web pages are image files, compressing bundles generally appears to provide no significant additional reduction in retrieval times. Our tests have also revealed that the time of availability of individual embedded objects and average delay per embedded object with bundle usage are equivalent to or better than those of burst-1.0 retrievals.

A key contribution of our work is a mechanism that gives Web servers control over the number and duration of TCP connections they support. We feel that our approach brings together the simplicity and determinism of the request-per-connection model and performance advantages of persistent connections. Our technique should be particularly useful for serving the most frequently accessed content at a site. The use of bundles allows small, relatively static, objects to be grouped to reduce requests and more efficiently use a single TCP connection. Its implementation requires no changes to the HTTP protocol. While its implementation does require changes to servers, proxies, and clients that wish to use it, these changes do not disrupt current practice. Bundle-aware clients will be able to use the mechanism as more servers support it without issues of selective feature support by clients and servers as occurs with HTTP/1.1.

Many directions for future work are possible. In our work we have conducted experiments to evaluate the bundle concept in its simplest form: combining all of the embedded objects in a container page into a single aggregate object that is served by the same machine as the container page. There are many other possible bundle creation and usage strategies that could be evaluated, such as bundling only the popular objects at a Web site, or serving a bundle from a content server rather than from the machine on which the container page is located.

We also need to conduct more work on the interaction between bundle usage and client-side caching. The first time a client retrieves a Web page and its embedded objects, retrieval of a bundle containing all of the embedded objects seems to be clearly advantageous. If the client caches these objects, retrieves other pages, and then later accesses the first page again, the client cache may contain some but not all of the objects in the original bundle. We have proposed solutions to this problem, but need to test them in the context of our prototype implementation.

Finally, we need to explore different compression and delta encoding algorithms for reducing the amount of content needed to be sent by a server to a client. We need to implement the *bsync* bundle-aware delta encoding mechanism. The idea of bundles could also be combined with prefetching as servers identify groups of objects that are most likely to be used. Finally we need to work on standardizing how bundles are computed along with how their existence is known to Web servers.

## 11 Acknowledgments

The authors thank Jeffrey Mogul and Balachander Krishnamurthy for comments on the original idea, and Mark Allman for answering questions regarding his work. The authors are also grateful to the National Science Foundation for providing partial support for this research.

## References

[1] Mark Allman. A Web Server's View of the Transport Layer. *Computer Communication Review*, 30(5), October 2000.

[2] Tim Berners-Lee, Roy T. Fielding, and Henrik Frystyk Nielsen. Hypertext Transfer Protocol—HTTP/1.0. RFC 1945, May 1996. `http://www.ietf.org/rfc/rfc1945.txt`.

[3] Scott Bradner. Key words for use in RFCs to indicate requirement levels. RFC 2119, IETF, March 1997. `http://www.ietf.org/rfc/rfc2119.txt`.

[4] K. C. Claffy, Greg J. Miller, and Kevin Thompson. The Nature of the Beast: Recent Traffic Measurements from an Internet Backbone. In *Proceedings of the INET '98 Conference*, Geneva, Switzerland, July 1998. Internet Society.

[5] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *ACM SIGCOMM'98 Conference*, September 1998.

[6] Theodore Faber, Joe Touch, and Wei Yue. The TIME-WAIT state in TCP and Its Effect on Busy Servers. In *Proceedings of the IEEE Infocom '99 Conference*, pages 1573–1583, New York, NY, March 1999. IEEE.

[7] Roy T. Fielding, James Gettys, Jeffrey C. Mogul, Henrik Frystyk Nielsen, Larry Masinter, Paul J. Leach, and Tim Berners-Lee. Hypertext Transfer Protocol—HTTP/1.1. RFC 2616, June 1999. `http://www.ietf.org/rfc/rfc2616.txt`.

[8] Jim Gettys and Henrik Frystyk Nielsen. Smux protocol specification. W3C Working Draft, July 1998. `http://www.w3.org/TR/WD-mux`.

[9] David Korn and Kiem-Phong Vo. The VCDIFF Generic Differencing and Compression Data Format. Internet Draft draft-korn-vcdiff-02.txt, November 2000.

[10] Balachander Krishnamurthy and Martin Arlitt. PRO-COW: protocol compliance on the Web. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, San Francisco, California, USA, March 2001. USENIX Association.

[11] Balachander Krishnamurthy, Jeffrey C. Mogul, and David M. Kristol. Key Differences Between HTTP/1.0 and HTTP/1.1. In *Eighth International World Wide Web Conference*, Toronto, Canada, May 1999.

[12] Balachander Krishnamurthy, Craig Wills, and Yin Zhang. Preliminary measurements on the effect of server adaptation for web content delivery. In *Proceedings of the ACM SIGCOMM Internet Measurement Workshop*, Marseille, France, November 2002. Short paper version.

[13] Balachander Krishnamurthy and Craig E. Wills. Piggyback server invalidation for proxy cache coherency. In *Seventh International World Wide Web Conference*, pages 185–193, Brisbane, Australia, April 1998.

[14] Balachander Krishnamurthy and Craig E. Wills. Analyzing Factors That Influence End-to-End Web Performance. In *Proceedings of the Ninth International World Wide Web Conference*, Amsterdam, Netherlands, April 2000.

[15] Media Metrix. `http://www.mediametrix.com`.

[16] Jeffrey Mogul. An alternative to explicit revocation?, Jan 1996.
`http://weeble.lut.ac.uk/lists/http-caching/0045.html`.

[17] Jeffrey Mogul, Balachander Krishnamurthy, Fred Douglis, Anja Feldmann, Yaron Goland, Arthur van Hoff, and Daniel Hellerstein. Delta encoding in HTTP, Oct 2000.
`http://search.ietf.org/internet-drafts/draft-mogul-http-delta-07.txt`.

[18] Jeffrey C. Mogul. The case for persistent-connection HTTP. In *Proceedings of the ACM SIGCOMM '95 Conference*. ACM, August 1995.

[19] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *ACM SIGCOMM'97 Conference*, September 1997.

[20] David Mosberger and Tai Jin. httperf – a tool for measuring web server performance. In *Workshop on Internet Server Performance*, Madison, Wisconsin USA, June 1998.

[21] Venkata N. Padmanabhan and Jeffrey C. Mogul. Improving HTTP latency. In *Second International World Wide Web Conference*, Chicago, Illinois, USA, October 1994.

[22] Jon Postel. Transmission Control Protocol. RFC 793, September 1981.
`http://www.ietf.org/rfc/rfc0793.txt`.

[23] National janet web caching service. Look for a local copy of this web site at the bottom of the page. `http://wwwcache.ja.net/`.

[24] Simon E. Spero. SCP—Session Control Protocol V 1.1.
`http://www.ibiblio.org/ses/scp.html`.

[25] W. Richard Stevens. *TCP/IP Illustrated, Volume 1, The Protocols*, volume 1. Addison-Wesley, Reading, MA, nov 1994.

[26] Andrew Tridgell and Paul Mackerras. The rsync algorithm, November 1998.
`http://rsync.samba.org/`.

[27] Greg Trott. Evaluating the effectiveness of bundles as a web content delivery mechanism. Master's thesis, Computer Science Department, Worcester Polytechnic Institute, May 2002.

[28] Zhe Wang and Pei Cao. Persistent Connection Behavior of Popular Browsers. Research Note, December 1998.
`http://www.cs.wisc.edu/~cao/papers/persistent-connection.html`.

[29] Duane Wessels. Squid Internet Object Cache. `http://www.squid-cache.org/`.

[30] Stephen Williams, Marc Abrams, Charles R. Standbridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proceedings of the ACM SIGCOMM Conference*, pages 293–305, August 1996.

[31] Craig E. Wills and Mikhail Mikhailov. Studying the Impact of More Complete Server Information on Web Caching. In *Proceedings of the 5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[32] Craig E. Wills, Mikhail Mikhailov, and Hao Shang. N for the price of 1: Bundling web objects for more efficient content delivery. In *Proceedings of the Tenth International World Wide Web Conference*, Hong Kong, May 2001.

[33] wwWhoosh, Inc. Optimizing web site performance. `http://www.wwwhoosh.com/products/download_docs/wwWhooshWhitePaper.pdf`.