Moving Edge-Side Includes to the Real Edge—the Clients

Michael Rabinovich† Zhen Xiao† Fred Douglis‡
Chuck Kalmanek†

†*AT&T Labs – Research*
‡*IBM T.J. Watson Research Center*

Abstract

Edge-Side Includes (ESI) is an open mark-up language that allows content providers to break their pages into fragments with individual caching characteristics. A page is reassembled from ESI fragments by a content delivery network (CDN) at an edge server, which selectively downloads from the origin content server only those fragments that are necessary (as opposed to the entire page). This is expected to reduce the load and bandwidth requirements of the content server.

This paper proposes an ESI-compliant approach in which page reconstruction occurs at the browser rather than the CDN. Unlike page assembly at the network edge, CSI optimizes content delivery over the last mile, which is where the true bottleneck often is. We call the client-based approach *Client-Side Includes*, or CSI.

1 Introduction

As the use of the Internet increases, caching is an important tool in coping with the rate of requests to Internet servers. Caching can be client-centric (proxy caching) or server-centric (reverse proxy caching or CDNs). Regardless, a significant limitation of caching is that it is mostly oriented toward *static* content. This limitation has been recognized and there have been a number of approaches to extend caching to handle other types of content.

These approaches include specialized tools to generate pages with dynamic content on a client (e.g., HPP [6] and <bigwig> [3]) or a proxy cache (e.g., Active Cache [4] and CONCA [19]); application distribution networks, which run complete applications at the edge of the network (e.g., Ejasent [9] and ACDN [13]); and Edge-Side Includes (ESI) [10], which builds pages, from component pieces (known as *fragments*) specified within an XML template, in servers at the network edge.

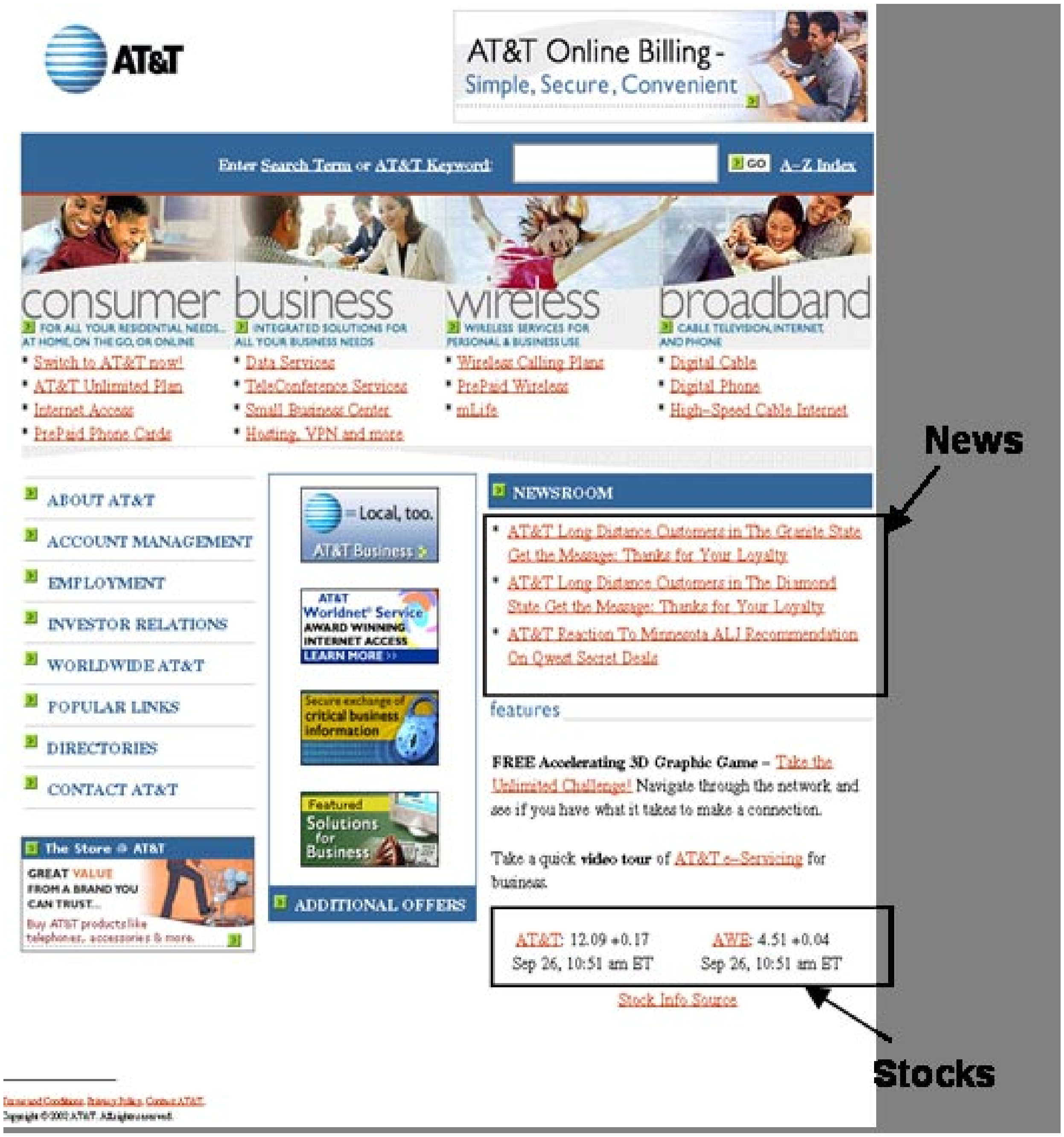
1.1 Fragment-based Technologies

As an example of fragment usage, consider AT&T's home page, `www.att.com`, shown in Figure 1. One can identify two natural fragments: one includes the newsroom headlines; another encompasses the stock prices for AT&T and AT&T Wireless and the date/time of these quotes. The rest of the page can be viewed as the template. The stock quote fragment changes very frequently, probably every minute when the stock market is open. The headlines fragments changes much more slowly, perhaps a few times each day. The rest of the page changes even less often.

If the entire page is considered as an indivisible whole, its lifetime in the cache is limited to the lifetime of the fastest-changing content, one minute in our example. With ESI, each portion of content is treated individually according to its own properties: the template will usually be accessed using the cached copy, the headlines fragment will be validated/refetched every few hours, and only the small stock quote fragment will be refetched every minute. We discuss ESI in greater detail in the next section.

The above example represents a typical organization of Web content, where the general "look and feel" of the page remains the same for a long time and only certain segments within this general page framework change. Several studies have found benefits from such page fragmentation [6, 21, 5]. The next question is where on the processing path from the origin server to the browser to reassemble a fragmented page? Historically, page assembly was first done at origin sites using technologies like Active Server Pages [1], Java Server Pages [11], PHP Hypertext Preprocessor [18], and Server-Side Includes [20]. The motivation behind this approach includes simplification of maintenance of Web sites (e.g., templates can be stored as static files and populated in a systematic way using data extracted from databases; different fragments can be conveniently generated by specialized application servers, etc).

Akamai implemented page reconstruction at the *edge*



Enter Search Term or AT&T Keyword: GO A–Z Index

consumer
FOR ALL YOUR RESIDENTIAL NEEDS... AT HOME, ON THE GO, OR ONLINE
- Switch to AT&T now!
- AT&T Unlimited Plan
- Internet Access
- PrePaid Phone Cards

business
INTEGRATED SOLUTIONS FOR ALL YOUR BUSINESS NEEDS
- Data Services
- TeleConference Services
- Small Business Center
- Hosting, VPN and more

wireless
WIRELESS SERVICES FOR PERSONAL & BUSINESS USE
- Wireless Calling Plans
- PrePaid Wireless
- mLife

broadband
CABLE TELEVISION, INTERNET, AND PHONE
- Digital Cable
- Digital Phone
- High–Speed Cable Internet

ABOUT AT&T
ACCOUNT MANAGEMENT
EMPLOYMENT
INVESTOR RELATIONS
WORLDWIDE AT&T
POPULAR LINKS
DIRECTORIES
CONTACT AT&T

The Store @ AT&T
GREAT VALUE FROM A BRAND YOU CAN TRUST...
Buy AT&T products like telephones, accessories & more.

= Local, too.
AT&T Business

AT&T Worldnet® Service
AWARD WINNING INTERNET ACCESS
LEARN MORE >>

Secure exchange of critical business information

Featured Solutions for Business

ADDITIONAL OFFERS

NEWSROOM
- AT&T Long Distance Customers in The Granite State Get the Message: Thanks for Your Loyalty
- AT&T Long Distance Customers in The Diamond State Get the Message: Thanks for Your Loyalty
- AT&T Reaction To Minnesota ALJ Recommendation On Qwest Secret Deals

features

FREE Accelerating 3D Graphic Game – Take the Unlimited Challenge! Navigate through the network and see if you have what it takes to make a connection.

Take a quick video tour of AT&T e–Servicing for business.

AT&T: 12.09 +0.17 Sep 26, 10:51 am ET   AWE: 4.51 +0.04 Sep 26, 10:51 am ET

Stock Info Source

Terms and Conditions, Privacy Policy, Contact AT&T
Copyright © 2002 AT&T. All rights reserved.

Figure 1: An example of a page amenable to ESI encoding, with two “fragments,” *news* and *stocks*, highlighted by boxes.

servers, outside the origin Web site, and ultimately coauthored the ESI language specification [8]. Other vendors supporting ESI include Speedera, a CDN that offers ESI page assembly at the edge similar to Akamai, and IBM, which offers edge servers supporting this functionality.

1.2 CSI: Addressing the Last Mile

ESI was proposed with the goal of assembling the page on surrogates: reverse proxies that act on behalf of the origin server in serving client requests, or edge servers in the CDN parlance. The ESI overview [10] claims that ESI "speeds up delivery of highly dynamic Web-based applications," in addition to the other goals of reductions in network and server loads. However, in this paper, we observe that page assembly at the edge server does not improve the response time for dial-up clients, which still represent a large majority of Web users (79% of consumer subscribers as of March 2002 [17]) and, while declining, are projected to remain a majority for the next several years (59% of all on-line households in the US in 2006 [12]). The reason for the lack of improvement in the dial-up environment is that the dial-up link (referred to as "the last mile") is often the bottleneck that determines the download time, and edge assembly does not affect the amount of content over that link. Furthermore, depending on the nature of content, ESI may actually increase the load on origin servers (see Section 4).

Consequently, we implemented an alternative mechanism, which performs ESI page assembly directly in the browser. We found that assembly in the browser can dramatically reduce the user response time. Depending on the nature of the page, we observed a significant reduction in the page display time for dial-up users with 56Kbps modems. Because page assembly occurs on the client, we call our approach *Client-Side Includes*, or *CSI*.

Our mechanism requires no modification of browsers or configuration of the browser machine. In particular, it does not require such typical extension techniques as configuring a new browser plug-in, or co-locating a custom proxy at the browser machine. At the same time, our mechanism implements most of the language specification of ESI 1.0 and thus enables content providers using ESI to switch to our mechanism without changing their content.[1]

An important advantage of CSI is that, unlike edge-based page assembly, CSI does not require the presence of an edge server. CSI can be implemented between the origin server and the browser directly. The existence of the edge server, and indeed the use of a CDN by the Web site, becomes an orthogonal issue. When a CDN is used, CSI can utilize edge servers for scalable delivery of page templates and fragments. Otherwise, browsers can download them directly from the origin server. This flexibility is important because inserting and removing ESI mark-up, however simple it might appear, has a high administrative overhead for large Web sites. With edge-side page assembly, a decision to use ESI entails a commitment for continued use of a CDN. With CSI, a Web site is free to use or not use a CDN based on other factors, such as performance and price.

Moreover, when the content provider does use a CDN, CSI can significantly reduce their CDN-related costs. The reason is that CDNs charge content providers for the traffic they deliver from their edge servers to clients. CSI reduces this traffic by delivering only ESI fragments, and not entire pages, from edge servers to clients.

It is important to realize the difference between a mark-up language and the mechanism for page assembly. Both our proposed CSI and the existing edge-side assembly mechanisms use the same mark-up language, called ESI. We will be careful to distinguish between the ESI *language* and the ESI *assembly mechanism* unless the meaning of the "ESI" term is clear from the context.

1.3 Our Contributions

While client-side assembly of a page from individual components has been described before [6, 3], our paper makes a number of contributions beyond prior work:

- While existing work used their own ad-hoc fragmentation languages, ours is to our knowledge the first paper that implements client-side page assembly using an existing widely supported language for page fragmentation, which was independently proposed for a different purpose. This is important because legacy content that already uses this language can benefit from our approach, whereas previous work required re-authoring the content.

- A significant concern with client-side implementations is the need to support the implementation across a vast number of different browser types and versions that co-exist on the Internet. We demonstrate that this concern can be effectively addressed without modifying the server code and without the need to maintain multiple versions of the content.

- To our knowledge, ours is the first paper that raises and addresses an issue of assessing whether and which pages should use fragmentation on a Web site.

- Finally, we provide details of our prototype implementation. While obviously dependent on the cur-

[1]Our current implementation excludes support for the optional "inlining" feature and provides incomplete support for ESI variables.

rent technology context, the lessons from our implementation experience will be useful for readers who implement other functionality on the clients.

2 ESI Overview

ESI is an open-standard XML-based markup language that provides a mechanism to assemble a web page from different components at the edge of a network. Each component can be retrieved independently and have its own cache control header, such as an expiration time. Since many web pages have a mixture of dynamic content (like stock quotes) and static content (like a page template), this design facilitates caching of web objects. Ideally, it helps reduce the congestion on the network, reduces the processing overhead on origin servers, and improves overall response time [10]. ESI has been developed by a consortium and subsequently proposed as a standard within the World Wide Web Consortium [8]. It is primarily targeted at processing on surrogates: reverse proxies that act on behalf of the origin server in serving client requests.

```
1. <HTML>
2. <!--esi
3.   <H3>Stock quote for ${QUERY_STRING}</H3>
4.   <esi:try>
5    <esi:attempt>
6.     <esi:include src=/quote.html
7.              alt=/delayed_quote.html/>
8.     <esi:choose>
9.       <esi:when
10.        test="$(HTTP_COOKIE{Type})==premium">
11.        <esi:include src=/market_news.html/>
12.      </esi:when>
13.      <esi:otherwise>
14.        To subscribe to premium services
15.         <A href=/subscribe.html> click here </A>
16.      </esi:otherwise>
17.    </esi:choose>
18.   </esi:attempt>
19.   <esi:except>
20.     <esi:include src=/sorry.html />
21.   </esi:except>
22.  </esi:try>
23. -->
24. <esi:remove>
25.    Please click on a
26.    <A href=/non_esi.html> non-ESI version </A> of this site
27. </esi:remove>
</HTML>
```

Figure 2: An example of ESI usage.

Figure 2 shows an example of an ESI template. The `<!--esi` and `esi:remove` tags allow templates that can be handled by non-ESI reverse proxies and browsers. If our template is obtained by a browser directly or through a non-ESI reverse proxy, the browser will assume that line 2 signifies the beginning of HTML comments and will ignore lines 3-21. The browser will further ignore unknown `esi:remove` tags and will display an invitation to access a non-ESI version of the site on lines 25-26. An ESI processor, conversely, removes `<!--esi` tags from the final page, and treats the text within the `esi:remove` block as comments. Thus, lines 3-21 will be processed and the invitation to access the non-ESI version will be elided.

Turning now to processing of lines 3-21, the `attempt` block of lines 5-18 is executed first. Line 6 tries to insert a fragment with relative URL "/quote.html". If for some reason this download fails (e.g., the data feed was broken), a fragment "delayed_quote.html" is tried next. If this download, or any other download in the attempt block, fails, the `except` block on lines 19-21 is executed.

Lines 8-17 give an example of a conditional inclusion based on testing a request cookie. Presumably the request Cookie header could contain "User-Type=premium" field, in which case line 11 would include the market news fragment, otherwise the invitation to subscribe to this service would be inserted into the final page. The `esi:remove` tag instructs the ESI processor to remove the enclosed text (the invitation to access non-ESI version of the site in our example) from the final page. The client that does not understand ESI would ignore the unknown tags and will display the text on lines 25-26.

This example illustrates fragment inclusion, conditional inclusion, and exception handling of ESI. Other features include a possibility of nested ESI constructs and access to environment variables. In particular, inclusion in ESI can be nested: the "/market_news.html" fragment in our example can itself include other fragments (or other ESI mark-up). Later versions of ESI also allow the template to declare and assign values to arbitrary variables, which can then be accessed (e.g., tested for conditional inclusion or inserted the assembled HTML page) inside ESI fragments.

3 Our Approach

There are a number of ways to assemble a page from various segments of information. The most common way is to do it at the origin server as shown in Figure 3a. With ESI, Akamai and others moved page assembly to CDNs' edge servers (Figure 3b). By reusing unchanged content from edge servers' caches, this approach reduces

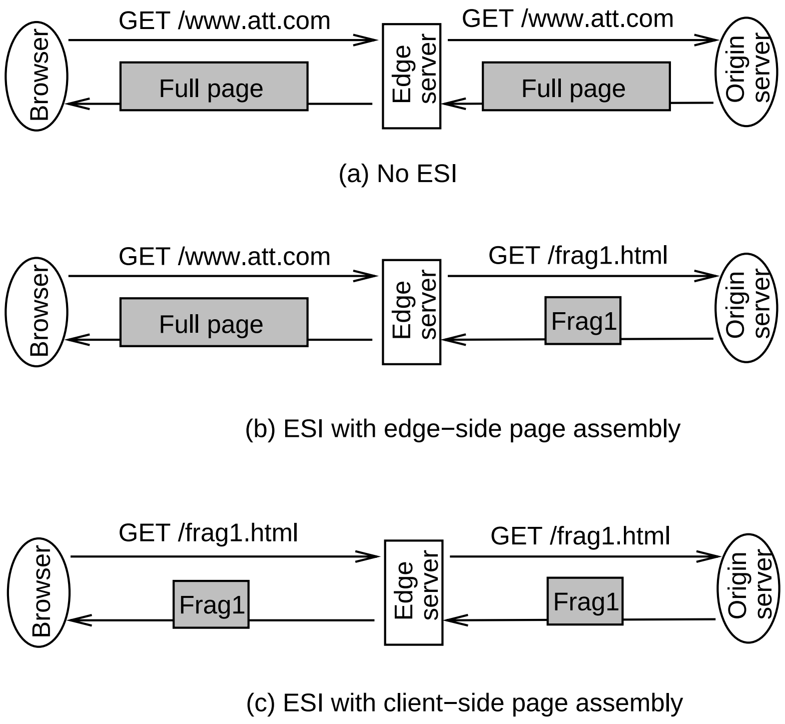


Figure 3: Page assembly alternatives assuming only fragment Frag1 must be prefetched into the cache.

traffic outflow from origin sites and associated connectivity costs for content providers. Also, depending on the nature of content, edge assembly may reduce server load (although it may also have an opposite effect as discussed in Section 4). By reducing bandwidth consumption and possibly server load, edge assembly can improve the peak capacity of origin sites. However, intuitively, edge assembly would not reduce download times for dial-up users during normal operation because the determining factor in this case is the slow dial-up link.

A separate study would be needed to conclusively verify this intuition. As a preliminary indication, we used a *wget* tool to download objects of around 100K from some remote sites (listed in Table 1) to a local idle server and compared the time it takes a dial-up client to fetch these objects from the remote sites and the local server. The client used a 56Kbps modem dialing into a modem bank that shared a building and a LAN with the local server. Thus, downloading from the local server emulated the best possible scenario of CDN-facilitated content delivery. As Table 2 shows, there is no discernible difference in median download times from remote and local servers except for the Wednesday experiment with Italy, which had two extremely high download times, 44 and 65 seconds.[2] We speculate that these two downloads reflect dropped connections by origin servers.

We propose to push page assembly further, to the ultimate network edge – the client itself. In our *CSI* approach, illustrated in Figure 3c, the client downloads only changed fragments, thereby reducing both the traffic that flows out of the origin server and that is transmitted over the last-mile link connecting the client to the Internet.

Just like edge assembly, CSI reduces connectivity costs for the content provider and possibly improves the origin site peak capacity. But it also improves the user experience for those with dial-up or other low-bandwidth connections. Furthermore, as already discussed in the Introduction, unlike edge assembly, CSI is applicable to Web sites regardless of whether or not they use CDN servers, and it reduces CDN-related costs for those Web sites that do use CDNs.

4 ESI or not ESI?

Before we explore different alternatives for ESI page assembly, an important question is how a content

[2]The large difference in download times between the three objects reflects the different degree of compressibility, since modems use hardware compression. As an indication of compressibility, the object size after *gzip* compression remained almost unchanged for the Italian object, reduced slightly to over 99K for the Cornell object, and reduced by more than the factor of 3, to just over 30K, for the Berkeley object.

Location	URL	Size
Italy	131.114.9.184/ luigi/rlc99.ps.gz	96983
Berkeley	169.229.60.105/ helenjw/papers/icc.ps	96176
Cornell	128.84.154.132/Info/Projects/Spinglass/public_pdfs/Randomized%20Error.pdf	115788

Table 1: Objects used in the comparison of download times. To factor out DNS latency, host names have been pre-resolved into IP addresses.

	Friday			Wednesday		
Object	local	remote	reduction	local	remote	reduction
Italy	17.5	17.8	2%	18.4	22.2	17.1%
Berkeley	8.8	8.7	-1%	9.3	9.7	4.1%
Cornell	19.9	19.9	0%	21.0	21.0	0%

Table 2: Median download times (in seconds) from remote sites and a local server. The table also shows the percentage of improvements due to proximity of the local server.

provider could decide if ESI encoding would be beneficial in its case. The answer depends on what the provider is trying to optimize.

- If the goal is to improve the end user experience, then ESI encoding is beneficial if it allows the content provider to move a substantial portion of content to templates or fragments with long TTLs. For poorly connected clients, the benefits are greatest if CSI assembly is used rather than ESI assembly, since traffic is reduced over the bottleneck link.

- If the goal is to reduce bandwidth consumption on the link from the Web server to the network, the condition that a substantial portion of content belong to ESI objects with long TTL will ensure that ESI is beneficial, regardless of the assembly method.

- Finally, if the goal is to increase the effective server capacity, the answer depends on the nature of the content. The following example elaborates upon this issue.

As an extreme example of a situation where ESI would be detrimental to server capacity, consider a 20K page that has three mutable regions of 2K each that change every minute. According to the ESI literature, this page seems like a good candidate for ESI encoding. However, for every HTTP request that reached the server without ESI, there will be either four requests (if it is a brand new client) or three (if is it a repeat client with a cached template). Overall, in the best-case scenario for ESI, when all requests are due to repeat clients, the bandwidth consumption out of the server reduces by a factor of three ($20K/(3 * 2K) = 3.3$), but the number of requests to the server increases three-fold.

If the bottleneck is the server load, the overall effect can easily be detrimental. Indeed, the server load generated by a request includes a fixed component that does not depend on response size and a variable component that is proportional to the response size. The relative contribution of these components depends on the nature of content (e.g., static files vs. CGI scripts in C vs. CGI scripts in Perl vs. fast CGI or servlets), and whether or not persistent connections are used. As one indication, we tested the request throughput of Apache 2.0.40 on a 733MHz NetBSD machine and, using static files with persistent connections, the throughput for a 20K page was reduced by only a factor of 2 as compared with the throughput for a 2K page – 1150 vs. 2274 requests per second.[3] So, in our example, with this workload, ESI would prove detrimental for effective server capacity even though it reduces the number of bytes served.

4.1 Guidelines

To generalize the above example, we propose the following procedure to infer the overall effect of reduced bandwidth consumption and increased request rate on the origin server capacity. We will use Figure 4 to illustrate the procedure, and we will explain various elements in this figure as we move along.

- Stress-test the server to see how its request throughput depends on the size of the response and plot the result. For example, the descending solid line on Figure 4 shows the throughput vs. response size curve obtained in our experiment with static files mentioned earlier. Call this a *capacity curve*. In obtaining the capacity curve, use resource types that approximate resources to be used in practice (e.g., Figure 4 was obtained using static files, which would be appropriate for sites like att.com, where

[3]A very similar trend can be seen in results by Nahum et al. for a variety of Web servers – see Table II in [16].

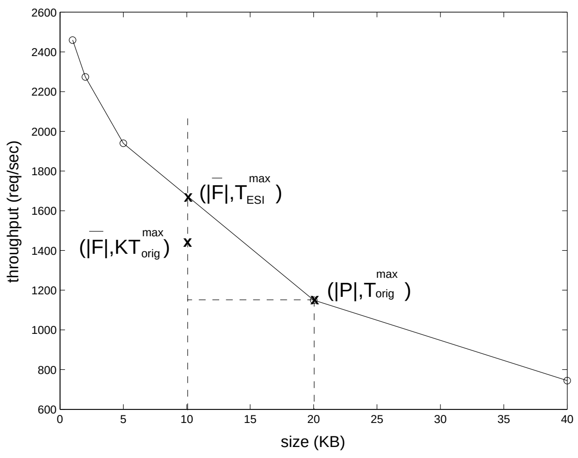


Figure 4: Effect of ESI encoding on server load.

even the most mutable fragments are only updated periodically; for other resources, servlets or CGI scripts might be more appropriate).

- Let $|P|$ be the size of the original page in question, that is, the page for which we must decide whether to use ESI encoding. Estimate (we will discuss how later) the average size of ESI objects, $|\bar{F}|$, that would be shipped from the server if the page were ESI-encoded and by how many times the request rate would increase, K. That is, if T_{orig} is the request rate for the original page, KT_{orig} will be the total request rate for the page template and fragments after it is ESI-encoded. Note that $|\bar{F}|$ is the average over server responses rather than the number of fragments on the page, so a more frequently requested fragment contributes more to the average.

- Draw two vertical lines in the throughput vs. object size plane, corresponding to the original and average ESI-encoded response size, $|P|$ and $|\bar{F}|$. For instance, the two dashed vertical lines in Figure 4 correspond to the case where the original page size is 20K and the average ESI-encoded response size will be 10K. Let $(|P|, T_{orig}^{max})$ and $|(\bar{F}|, T_{ESI}^{max})$ be the points where these vertical lines intersect with the capacity curve. T_{orig}^{max} represents the server capacity for the original page and T_{ESI}^{max} gives the server capacity for the ESI-encoded page.

- We know that, whatever the request rate for the original page was, the request rate for the ESI objects after ESI encoding will be K times higher. In particular, the maximum sustainable demand for the original page (corresponding to T_{orig}^{max} request rate) results in the request rate for the ESI objects that is equal to $T_{after} = K \times T_{orig}^{max}$. If the request rate $T_{after} = T_{ESI}^{max}$, then the server will remain fully utilized after ESI-encoding, and its effective capacity will remain the same. If $T_{after} < T_{ESI}^{max}$, the server utilization will be below capacity and hence it could serve some additional requests. In other words, ESI encoding would increase the effective capacity of the server. Finally, if $T_{after} > T_{ESI}^{max}$, ESI encoding will reduce the effective capacity. Graphically, we plot the point $(|(\bar{F}|, K \times T_{orig}^{max})$ on the coordinate plane. If this point is above the throughput curve, ESI encoding of page P will reduce the effective server capacity, otherwise the capacity will increase. For example, in Figure 4, T_{orig}^{max} is 1150 requests per second. Assuming $K = 1.25$, the same demand that drove the server to capacity would result in $1150 \times 1.25 = 1437.5$ requests per second for ESI objects, which falls below the capacity curve and thus indicates that ESI encoding of this page would be beneficial from the perspective of server capacity.

4.2 Estimating $|\bar{F}|$ and K

To estimate $|\bar{F}|$ and K, the most general technique is a trace simulation using the Web server access log. The simulator should translate each request for page P in the log into requests for the template and every fragment, and then filter out repeated requests from the same client that would hit in the client's cache, taking into account TTLs of individual ESI objects. The remaining requests could then be used to calculate $|\bar{F}|$ and the new request rate.

However, some special cases, such as the case where every fragment has a non-zero TTL, allow analytical estimations. Consider a Web site that uses a CDN and a page P of size $|P|$ that contains n fragments, $F_1, ..., F_n$, with each fragment F_i having a lifetime of t_i seconds and size $|F_i|$. Let t_m be the smallest lifetime of all fragments, with t_m greater than 0.

With ESI, in the best case of all repeat clients, the template is (almost) never sent. Assume that page P is popular enough so that the request rate for it at each of the CDN's edge servers is much higher than fragment lifetimes. (After all, unpopular pages do not matter from the perspective of load.) Then, without ESI, an edge server will send $1/t_m$ requests per second. With ESI, the edge server will send $\sum_{i=1}^{n}(1/t_i)$ requests per second.

Since the same is true for all edge servers, the request rate is increased by a factor of

$$K = [\sum_{i=1}^{n}(1/t_i)]t_m$$

```
<HTML>
  <BODY>
    <SCRIPT SRC="csi.js"> </SCRIPT>
    <SCRIPT> run("page_template.html"); </SCRIPT>
  </BODY>
</HTML>
```

Figure 6: The wrapper for Javascript/ActiveX implementations of CSI.

.

Similarly, the amount of data served per second to the edge server is $\sum_{i=1}^{n}(|F_i|/t_i)$ and therefore the average response size is

$$|\bar{F}| = \frac{\sum_{i=1}^{n}(|F_i|/t_i)}{\sum_{i=1}^{n}(1/t_i)}.$$

5 Implementation

CSI is a mechanism for assembling a page from individual ESI components at the browser. Any implementation of CSI must be able to download page components, process them to assemble the page, and tell the browser to display the result. We implemented CSI using JavaScript for assembling the page. Our implementation follows the framework shown in Figure 5. When a CSI-capable client requests a page, the server returns a small wrapper (150 bytes plus headers). The wrapper invokes a Javascript page assembler and passes it the URL of the ESI template corresponding to the requested page. The page assembler then downloads the template and any ESI fragments it includes and assembles the page.

It might appear that CSI involves much overhead to download the page assembler script and the page wrapper. However, the assembler script is generic for all CSI content from any Web site. Thus, once a client downloads it, it remains in its cache and is invoked locally. This is akin to installing a piece of software on the client except this software is installed transparently the first time it is used. The wrapper is immutable for a given page but not across pages because it includes the URL of the requested page. Thus, the client does incur an overhead of fetching the wrapper when it accesses the page for the first time. Subsequent accesses to this page will not incur this overhead because they will reuse the cached wrapper even if the page itself has changed. Because the wrapper is very small, the above overhead is mostly due to round-trip packet latency and not bandwidth consumption.

We implemented CSI for Microsoft's Internet Explorer browser using ActiveX to download page components. In principle, Java's LiveConnect facility can be used instead, except we in the past encountered incomplete support of this feature in MSIE [7]. The implementation uses the wrapper shown in Figure 6. In this wrapper, the csi.js script implements the page assembler, and `run` is the function in the assembler that starts the processing. The assembler downloads page components using the code fragment below:

```
httpDoc = new ActiveXObject("Microsoft.XMLHTTP");
httpDoc.open("GET", url, false);
httpDoc.send();
if (httpDoc.status != 200) <Process exception>
```

This implementation requires that ActiveX be enabled in the browser. Although this is a default configuration for MSIE, some users disable ActiveX due to security concerns, in which case we resort to a fall back approach that we also use for non-IE browsers. We discuss this fall-back approach next.

5.1 Supporting Non-IE Browsers

Our implementation of CSI only works for Microsoft Internet Explorer (MSIE). Although MSIE now occupies the overwhelming majority of the browser market, an important issue we need to address is how to deal with non-MSIE browsers and various early versions of the MSIE browsers that might not be compatible with the Javascript or ActiveX features used by our implementation. One could attempt to support different CSI implementations for all possible browsers. This would be an administrative nightmare and a significant disadvantage over edge assembly of ESI pages which is browser-agnostic. Alternatively, a Web site could maintain two versions of the content, one with ESI markups for CSI-capable clients and the other for other clients. However, maintaining the content in two versions is unacceptable to many content providers due to administrative costs.

Our approach to this problem is to implement CSI for the prevalent browser only (recent versions of MSIE) and to resort to edge-side or server-side page assembly for all other browsers. This approach can be implemented in two ways, which decide between CSI and ESI either at the client or at the server.

The client-side solution adds a test for browser capabilities to the wrapper and redirects the non-CSI capable browser to the ESI script with the template URL as a parameter. Figure 7 shows an example of such wrapper for the ActiveX CSI implementation. If the browser does not support Javascript, the wrapper invites the user to click for the ESI script. Although awkward, this seems acceptable as Javascript is virtually universally enabled. The disadvantage of the client-side solution is that it increases the size of the wrapper.

The server-side solution is enabled by HTTP's User-agent header, which nearly all browsers include with

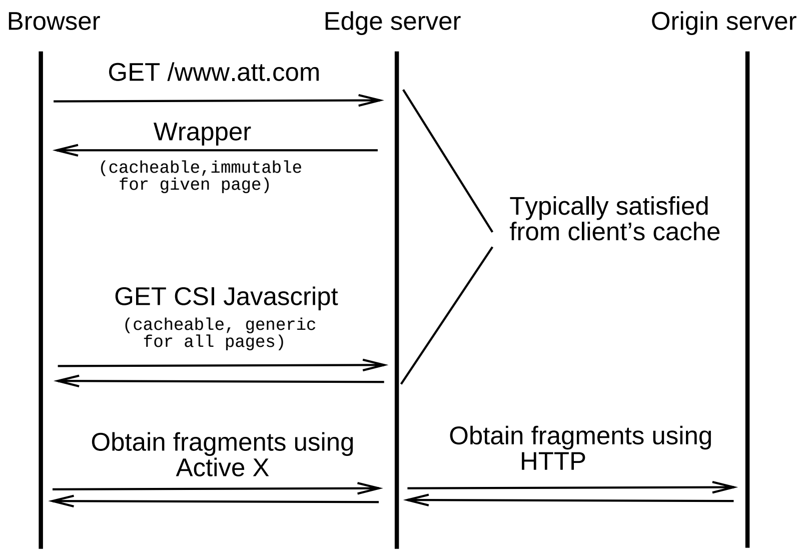

Browser
Edge server
Origin server
GET /www.att.com
Wrapper
(cacheable,immutable for given page)
Typically satisfied from client's cache
GET CSI Javascript
(cacheable, generic for all pages)
Obtain fragments using Active X
Obtain fragments using HTTP

Figure 5: CSI interactions for a browser that never before accessed any CSI-enabled page.

```
<HTML>
  <BODY>
    <SCRIPT>
    <! – –
      if (!window.ActiveXObject)
        window.location="/cgi-bin/esi.pl/template.html"
    //- - >
    </SCRIPT>
    <SCRIPT SRC="csi.js"> </SCRIPT>
    <SCRIPT>
    <!- -
      run("template.xml");
    //- - >
    </SCRIPT>
    If your browser does not support Javascript please click
    <A href="/cgi-bin/esi.pl/template.html"> here </A>
  </BODY>
</HTML>
```

Figure 7: The wrapper choosing between client- and server-side page assembly.

their requests. If the server processing the request (the origin server or edge server if CDN is used) can determine from this header that the client is CSI-capable, the server returns the CSI wrapper. Otherwise, the server reconstructs the page itself and returns the complete HTML page. This client-specific request processing can be done without any modification of server code. On Apache, it can be achieved by an appropriate server configuration. One configuration method, which we tested, uses Apache's URL rewriting [14]. Here, a requested URL is rewritten into different internal URLs depending on the value of the User-agent field in the HTTP request header. The disadvantage of the server-side solution is that it does not work well with client Web proxies. The server now returns different responses for the same request based on the nature of the browser. If a proxy has both CSI-capable and CSI-incapable clients, the proxy should also distinguish between these responses and not send a CSI response to a CSI-incapable browser. The server can ensure this behavior by adding "Vary: User-agent" HTTP header to its responses. Unfortunately, some proxies do not cache responses with this header, while those that do will not share a cached response among *any* non-identical browsers even if all of them support CSI (such as different versions of MSIE). In either case, the effectiveness of proxy caching would be reduced.

5.2 Discussion

There are a number of other ways to implement the CSI functionality. Possible approaches include using pure Java, Javascript/Java combinations; pure Javascript; local proxy caches; and XML with XSLT transformations.

In the pure Java approach, the exposed URL of an ESI-encoded page would return a small wrapper object, which would invoke the applet that implements CSI, passing to this applet the URL of the template. The applet would fetch the template and assemble the page, downloading page fragments as needed, and then display the page. The applet itself would be generic for any ESI-encoded page and would typically be found in the browser cache. The Javascript/Java combination would be similar, except the wrapper would invoke a Javascript module rather than the applet, page parsing and assembly would occur in that Javascript module, and Java would only be used to download the template and fragments by means of the Java's LiveConnect facility. We chose ActiveX over Java in our initial implementation because the former is better supported in MSIE. Because MSIE constitutes the vast majority of browsers, any performance optimization must apply to this browser in order to have any practical effect.

Pages could be generated on a client machine in a separate application, as is done with CONCA [19]. This approach requires explicit action on the part of users, and is therefore not suited to CSI in its current form.

A subset of ESI functionality can be implemented using XML and XSLT. The XML/XSLT implementation would treat ESI tags as XML elements and provide XSL procedures to process these elements by replacing them with appropriate ESI fragments. The XML/XSLT implementation does not require a separate wrapper. Another advantage of this implementation is that XML/XSLT is being adopted across all browsers. A disadvantage is that does not have access to HTTP-related variables such as request header fields. So, it can only implement a subset of the language. Given all other options, this implementation approach does not appear compelling.

6 Performance

In this section, we measure the performance of ESI/CSI. The measurements were conducted using a set of synthetic pages and two real pages. The synthetic pages were HTML files of specified sizes with randomly generated characters. When ESI encoding is used, these pages are split into a template and a specified number of fragments. This allows us to study performance trends of ESI/CSI in a systematic manner by varying the degree of caching for the CSI JavaScript, the template, and the fragments. In this experiment, we generated synthetic pages of 20K, 60K, and 100K bytes. Each page consists of a template and four fragments. The template has 80% of bytes in the page, and each fragment has 5%. As a target for comparison, we also generated a static page for each page size by assembling the template and its fragments offline.

For the real pages, we chose AT&T's entry page, `http://www.att.com`, and the Wall Street Journal's entry page, `http://online.wsj.com/`. We downloaded a copy of each page and its embedded objects to a local Apache server and then manually split the page into an ESI template and a set of fragments. The AT&T page has two fragments (as shown in Figure 1): news headlines and stock quotes. The Wall Street Journal page has three fragments: time of the day, news headlines, and stock quotes.

Page	Static page	with CSI	with ESI
synthetic 20K	240	320	380
synthetic 60K	300	431	441
synthetic 100K	441	491	501
AT&T page	306	351	430
WSJ page	571	676	831

Table 3: Overhead of CSI and ESI processing. All numbers are in milliseconds.

The server used in the experiments runs on a 864Hz Pentium III with 256 MBytes of memory. The client computer is an IBM T22 Thinkpad laptop with a 1GHz CPU and 128MB memory running Windows 2000.

6.1 Overhead

We first measure the overhead of ESI and CSI processing. To do so, we compare the display time of an ESI-encoded page using server-side assembly or client-side assembly with that of the corresponding static page. We implemented server-side assembly as a Perl script with FastCGI. Since our software resorts to server-side assembly for clients considered CSI-incapable, we want to make sure the overhead is acceptable. The experiment was conducted over an 100Mbps Ethernet and was repeated 30 times. In repeat downloads, all page components were fetched from the browser cache. Table 3 shows the median display time for different synthetic pages as well as for the two real pages. All numbers are in milliseconds.

As can be seen from the table, the overhead for server-side assembly is small: below 300ms in all cases. This overhead can be significantly reduced by reimplementing page assembly in C. However, given that only very few clients will experience it, we consider it to be already acceptable. We should also note that in reality these pages would not be downloaded from static files anyway. Rather, they need to be assembled on the server using some mechanism such as server-side includes. Thus, our measurements provide an upper bound of the overhead.

The table also indicates that the overhead for CSI assembly is under 150ms. As we will see in the next subsection, this overhead is more than offset by savings in transfer times for dial-up clients. While for broadband clients it might result in a slight increase of the overall download time, we believe it is justified by a significant reduction in download time for dial-up clients.

6.2 Display Time

Next we measure the time it takes for a client to retrieve a page from the Web server and display it in its

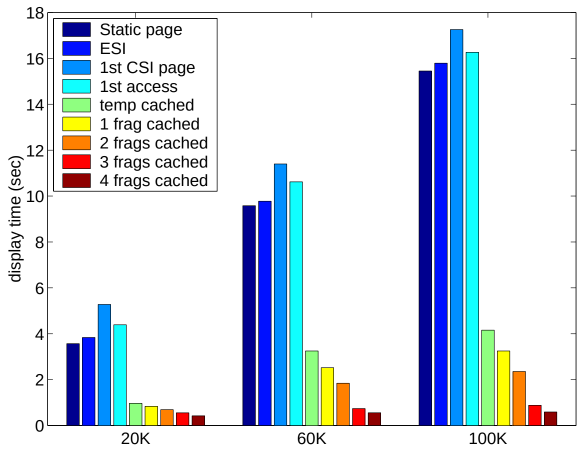


Figure 8: Download time of synthetic pages over dial-up links.

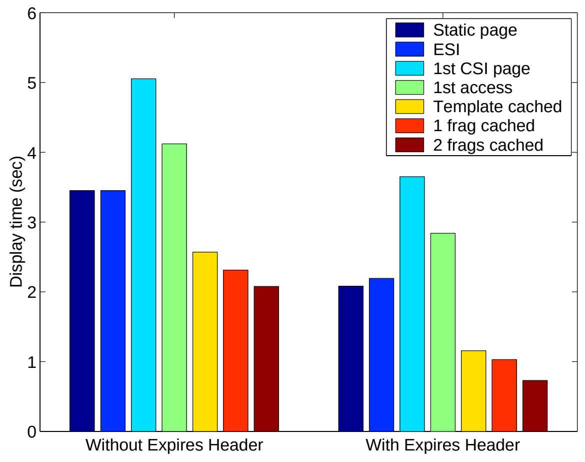


Figure 9: Download time of the AT&T entry page over dial-up links.

browser over dial-up links with 56K modems. The results for synthetic pages are shown in Figure 8. The figure indicates that the display time for ESI is slightly higher than that of the static page due to the processing overhead of the ESI Perl script. The figure also shows that there is a substantial penalty in performance when a client invokes CSI for the first time (denoted as "1st CSI page" in the figure). In this case, the browser needs to download the CSI JavaScript, the wrapper page, the template, and all the fragments. For 20K pages, it increases the display time by as much as 48%. Since the CSI script is the same for all pages, it can be served from the browser cache afterward. Even so, the first access to an CSI-enabled page (denoted as "1st access") may still incur an overhead due to the download of the wrapper page and the processing of CSI script inside the client's browser. However, during subsequent visits to the page, the template is likely to be served out of the cache since it seldom changes. Consequently, only those fragments that have changed need to be fetched from the server. As can be seen from the figure, this results in substantial reduction in display times across all page sizes.

The results for the AT&T page are shown in Figure 9. The figure indicates that CSI improved the median download time by about 25%, from 3450ms to 2569ms, assuming that the template is cached, which would be the typical case. Note that the performance improvement for the AT&T page is not as substantial as that for synthetic pages. We discovered that this is because the AT&T page is more compressible than randomly generated synthetic pages.

We further observed that the AT&T site (like many other Web sites) does not specify expiration times for its embedded objects. MSIE in this case sends "If-Modified-Since" requests to validate these objects on each access. Since these objects (mostly images) do not change often, we configured the local server to provide expiration times for them. Then we repeated the experiments to measure the download time for what could be considered a "properly configured" Web site. With explicit expiration times, all values reduced accordingly and CSI's relative improvement grew to 45%. Like in synthetic pages, the first access to the AT&T page may have a high overhead due to the download of the wrapper page, the template, and all the fragments.

Figure 10 shows the results for the Wall Street Journal page. When the template was served from the browser cache, CSI reduces the display time by about 27%. If expiration headers were provided for the embeded objects, the improvement becomes 38%. Note that some of these objects have query strings in their URLs. This causes MSIE to send validation requests even if they have not expired.

6.3 Bandwidth Reduction

An important goal of our project is to reduce the amount of bytes that need to be transmitted over the last mile. Table 4 shows the sizes of the two real pages and their components with ESI-encoding.[4] The table indicates that 93% of the bytes in the AT&T page and 71% of the bytes in the Wall Street Journal page are in their templates. Hence, client-side assembly of these pages can achieve significant reduction in bandwidth when the templates are in the browser's cache. Note that the ac-

[4]The total size of the page template plus all its fragments is slightly higher than the size of the full page. This is due to some ESI encoding statements we added into the page.

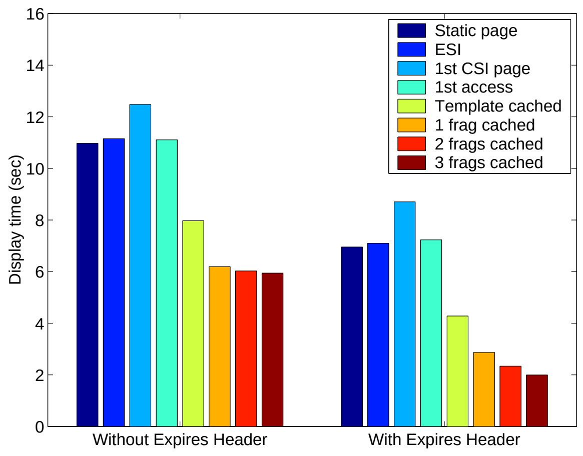


Figure 10: Download time of the Wall Street Journal entry page over dial-up links.

	AT&T Page	WSJ Page
full page	30731 (100%)	79608 (100%)
page template	28661 (93%)	56324 (71%)
current time	N/A	55 (0%)
news headlines	927 (3%)	20161 (25%)
stock quotes	1231 (4%)	3166 (4%)

Table 4: Sizes of the AT&T page and the Wall Street Journal page with ESI encoding.

tual saving observed in practice is slightly lower than indicated in the table because of the extra bandwidth consumed by HTTP requests for the page components and by HTTP headers carried by responses with the components. We estimate that this adds about 400 bytes for each fragment that is not in the cache.

7 Limitations and Future Work

The need to download the wrapper increases latency when the browser accesses the page for the first time and hence does not have the wrapper in its cache. Our particular implementation downloads fragments into the CSI assembler script sequentially and synchronously with template parsing. This may slow down page assembly for pages containing a large number of fragments that are not locally cached. Furthermore, MSIE seems to always validate any locally cached object that does not have an explicit expiration time, and in our case these fragment validations would be sequential. Thus, CSI will bring higher benefits to those Web sites that supply Expires headers for their fragments.

A limitation not specific to our implementation stems from the fact that Javascript is allowed to transparently download objects only from the same Web site from which the original page (that invoked the Javascript) was downloaded. In other case, this means that the template and all fragments must come from the same Web site. This disallows fragment sharing among Web sites. Edge-side assembly is not restricted by existing browser realities and therefore allows fragment sharing.

Some pages that are well suited to ESI assembly may not be amenable to CSI. Specifically, pages that are accessed by very many clients, but only once per client over a long interval, may be generated efficiently within a CDN but slow down individual CSI clients due to the "first access" downloads discussed in the previous section.

As future work, we would like to extend the ESI language with some features from the HPP markup language [6], such as the loop construct. This would expand the applicability of ESI to more content, most notably responses from search engines.

8 Related Work

Numerous language constructs exist for including fragments into an HTML page. IMG and APPLET HTML tags tell the browser to insert, respectively, an image or an applet to the HTML document. The OBJECT tag allows an insertion of an object of an arbitrary type. These tags, however, only allow a straightforward inclusion. In particular, they allow no conditional inclusion, no access to environment variables and HTTP headers, and no user-defined variables to pass data from the containing page to included objects. The same is true of the Xinclude construct in XML. The ESI language allows much greater flexibility in specifying how the final page should be assembled.

Much more sophisticated inclusion mechanisms than the above-mentioned tags exist for use on the server-side. ASP, JSP, PHP, and Server-Side Includes are pure server-side tools. Unlike CSI, the page using these language constructs is assembled at the server and shipped to the client in its entirety. Thus, the client only sees the final HTML page without any inclusion constructs. The primary goal of these tools is to simplify the development and management of the Web site (for example, to provide a systematic way of organizing a database-driven Web site), and not to improve the performance of accessing the site. The purpose of the ESI language, and the CSI assembly mechanism in particular, is to improve performance.

The closest approach to CSI are HTML Pre-Processing (HPP) [6] and the <bigwig> project [3]. HPP is geared towards the case where a page contains access-specific information that changes on every access. In particular, HPP distinguishes only two kinds of content on the page - the static template and (possibly disjoint) dynamic portions that must always be downloaded from the server. The ESI language allows the containing page to specify multiple fragments, each with its own caching characteristics. At the same time, some language features from HPP, most notably its loop construct, would benefit the ESI language. In terms of page reassembly, HPP was implemented as a browser plug-in, thus requiring browser configuration. In contrast, CSI requires no such configuration. <bigwig> is similar to CSI in that it uses Javascript for page reconstruction on the client. However, it is based on its own language for Web service specification, and is applicable only to applications created with that language.

Active Cache [4] and CONCA [19] are two examples of dynamic content generation within intermediaries such as proxy caches. Unlike ESI page assembly, which uses surrogates associated with the content provider, these systems permit dynamic content to be generated closer to end users, and possibly under their control. In particular, CONCA permits user profiles to assist with transcoding content formats to a user's specifications. With CSI, our emphasis is to generate content dynamically with an established language (the ESI language) and existing browser technologies, Javascript and ActiveX.

A recently emerged notion of "utility computing" aims at providing an even higher degree of flexibility in page construction. ACDN [13] and Vmatrix [2] are examples of research efforts, and Ejasent [9] is one start-up commercial venture in this area. Rather than providing a mark-up language to insert dynamic fragments, the idea of utility computing is to allow entire applications to run at CDN servers and to let these applications migrate or be replicated among CDN servers as needed by changing demand. Utility computing is a pure edge-side approach. It aims at replicating applications rather than optimizing data transfer over the last mile. Thus, while this technology does overlap with edge-side page assembly, it is complimentary to CSI.

Finally, like previous systems like HPP and Conca, the benefits of CSI should be evaluated relative to another emerging technology, delta-encoding [15]. Sending updates to web pages, rather than the pages themselves, can save bandwidth and improve response time; it can be deployed over the last mile, between the user and a proxy; between a proxy (including a CDN) and a content provider; or across the entire connection. For the last mile, CSI has the advantage of transparency, since there need be no special support in the browser or a local proxy. It also does not require synchronization between clients and servers to specify base versions against which to compute deltas, and does not require the same content to be transmitted for different pages [19].

9 Conclusions

Numerous methods for template-based page construction exist. These methods are typically oriented toward simplifying content generation and management are intended to be executed at the origin site before the page is shipped out. The ESI language was proposed with the goal of shifting template-based page assembly to the network edge. This paper argues for shifting such page assembly all the way to the browser and shows that such a shift can occur transparently to browsers, while supporting most of the ESI language.

Our proposed client-side page assembly (CSI) not only reduces the traffic served by origin servers, it also reduces the traffic flowing into the client over the last-mile connection. In the case of dial-up clients, this translates into a significant reduction in end-user response times. Furthermore, for content providers who use CDN services, CSI can significantly reduce their CDN-related costs by delivering only ESI fragments, and not entire pages, from edge servers to clients.

Our performance study shows that the overhead of CSI assembly is small and is more than offset by the reduction in transmission time over the last mile. Our micro-benchmark pages, as well as experiments with two sample real pages, showed very significant net reductions in page time-to-display. Edge-side page assembly overhead, while similar to CSI, occurs *in addition* to the last mile transmission time. It therefore only adds to the total time-of-display of ESI pages, assuming that the last mile is the bottleneck.

Acknowlegements

We wish to thanks Walter Sturm for encouragment to pursue this project and to Irina Rabinovich for help with our Javascript and XML questions. We also thank anonymous referees for helpful comments.

References

[1] Guide to active server pages. `http://msdn.microsoft.com/library/default.asp?URL=/library/psdk/iisref/aspguide.htm`.

[2] Amr Awadallah and Mendel Rosenblum. The vMatrix: A network of virtual machine monitors for dynamic content distribution. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, 2002.

[3] C. Brabrand, A. Møller, S. Olesen, and M.I. Schwartzbach. Language-based caching of dynamically generated HTML. *World Wide Web*, 5(4):305–323, 2002.

[4] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the Web. In *Proceedings of the 1998 Middleware Conference*, September 1998.

[5] Jim Challenger, Arun Iyengar, Karen Witting, Cameron Ferstat, and Paul Reed. A publishing system for efficiently creating dynamic web content. In *Proceedings of INFOCOM*, pages 844–853, 2000.

[6] Fred Douglis, Antonio Haro, and Michael Rabinovich. HPP: HTML macro-preprocessing to support dynamic document caching. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, pages 83–94, December 1997.

[7] Fred Douglis, Sonia Jain, John Klensin, and Michael Rabinovich. Click-once hypertext: Now you see it, now you don't. In *Proceedings of the Second IEEE Workshop on Internet Applications*. IEEE, July 2001.

[8] Edge Side Includes W3C submission. `http://www.w3.org/Submission/2001/09/`, September 2001.

[9] Ejasent, inc. `http://www.ejasent.com`.

[10] ESI - Accelerating E-Business Applications: Overview. `http://www.esi.org/overview.html`, September 2002.

[11] JSP: Java server pages. `http://java.sun.com/products/jsp/`.

[12] Jupiter internet access model (US only). Broadband. Jupiter Media Metrix, August 2001.

[13] P. Karbhari, M. Rabinovich, Z. Xiao, and F. Douglis. ACDN: a content delivery network for applications. In *Proceedings of the ACM SIGMOD Conference, Demonstrations track*, page 619, 2002.

[14] Module mod_rewrite URL rewriting engine. `http://httpd.apache.org/docs/mod/mod_rewrite.html`.

[15] Jeffrey Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *Proceedings of the ACM SIGCOMM Conference*, pages 181–194, September 1997.

[16] Erich M. Nahum, Tsipora Barzilai, and Dilip Kandlur. Performance issues in WWW servers. *IEEE/ACM Transactions on Networking*, 10(2):2–11, Feb 2002.

[17] Network traffic & revenue analysis. market update. RHK, Inc., July 2002.

[18] PHP: Hypertext preprocessor. `http://www.php.net/`.

[19] W. Shi and V. Karamcheti. CONCA: An architecture for consistent nomadic content access. In *Workshop on Cache, Coherence, and Consistency (WC3'01)*, June 2001.

[20] Apache tutorial: Introduction to server side includes. `http://httpd.apache.org/docs/howto/ssi.html`.

[21] Craig E. Wills and Mikhail Mikhailov. Examining the cacheability of user-requested Web resources. In *Proceedings of the 4rd Web Caching and Content Delivery Workshop*, April 1999. http://workshop99.ircache.net/Papers/wills-final.ps.gz.