Enabling Network Caching of Dynamic Web Objects

Pankaj K. Garg, Kave Eshghi
HP Labs

Thomas Gschwind
Technische Universität Wien

Boudewijn Haverkort
RWTH Aachen

Katinka Wolter
Technische Universitä Berlin

April 11, 2002

Abstract

The World Wide Web is an important infrastructure for enabling modern information-rich applications. Businesses can lose value due to lack of timely employee communication, poor employee coordination, or poor brand image with slow or unresponsive web applications. In this paper, we analyze the responsiveness of an Intranet web application, i.e., an application within the corporate firewalls. Using a new Web monitoring tool called WebMon, we found, contrary to our initial expectations, substantial variations in the responsiveness for different users of the Intranet Web application. As in the Internet, traditional caching approaches could improve the responsiveness of the Intranet web-application, as far as static objects are concerned. We provide a solution to enable network caching of *dynamic* web objects, which ordinarily would not be cached by clients and proxies. Overall, our solution significantly improved the performance of the web application and reduced the variance in the response times by three orders of magnitude. Our cache enabling architecture can be used in other web applications.

Keywords: Software Performance Monitoring, Web Performance Tuning, Response Time Breakdown, Application Response Monitoring, Web Caching, Dynamic Web Pages, Cache Update Policies.

1 Introduction

Since its inception in the early 1990's, the World Wide Web has evolved into being the most widespread infrastructure for Internet-based information systems. As with traditional information systems, the response time (measured starting from the user's initiation of a request and ending when the complete response is returned to the user) for individual interactions becomes critical in providing the right Quality of Experience (QoE) for a web user. A fast, responsive application gives an impression of a well-designed and well-managed business and web service. A slow, un-responsive application, on the other hand, can suggest poor quality of business and may send users seeking alternative web sites or services [**?**].

Unlike traditional information systems, however, not all components required to accomplish a web transaction are necessarily under the control of the organization running the web application. Typically, end-users access a web application with a web browser, through a collection of proxies or caches, which ultimately send the request to a back-end web server farm with application servers and databases. Several networks participate in the intervening layers. Such *organizational heterogeneity*, in addition to the *component heterogeneity*, complicates the designs for responsiveness of web applications.

Caching of web objects has emerged as key enabler for achieving fast, responsive web applications [**?**]. The main idea is to maintain copies of web objects close to the end-user's browser and serve requests from caches to improve latency and reduce effects of queueing in network elements along the path from the browser to the server. Web browsers maintain cached copies of frequently requested objects. Similarly, large organizations or Internet Service Providers (ISPs) maintain caches of web objects frequently requested and shared by their user community.

To fully utilize the benefits on the Internet or Intranet, effective caching requires web objects available as **static** HTML pages. This means that the HTML page for that web object was not generated when a user requested the object, but was available as a stored HTML file. This is important because the intervening caches, e.g., the browser's cache, will normally not cache **dynamic** content but it will cache static content. Approaches for enabling caching of dynamic content through HTTP headers, e.g., with the `Expires` header, although a step in the right direction, are not universally and uniformly honored by intervening caches.

For performance reasons, web site administrators are well advised to turn dynamic content into static content whenever necessary and possible, to avoid the above problems.

In this paper we provide a framework for: (1) monitoring web applications performance to determine the need for a higher static-to-dynamic web-object ratio, and (2) an architecture to turn dynamic content into static content based on existing file systems and dependency tracking tools, e.g., using the well-known `make` facility [?]. Our approach relies on the fact that some dynamic objects do not change that often, and the data required to construct them is available in files rather than in a database.

We use WebMon [?] to monitor a web application's performance, which determine the requirements for enabling network caching of dynamic content. WebMon monitors the end-user perceived response time for web transactions and reports to a central server. At the same time, WebMon correlates client perceived response times with the server response time for each transaction. Statistical analysis of such client and server response times will help in understanding the requirements with respect to caching. Once a caching solution has been implemented, WebMon can again be used to quantify the benefits of caching.

When we turn dynamic content into static content, the user's request does not usually come all the way to the web server, but may be served by an intermediate cache. Although good for performance reasons, this is a drawback if the web server's administrator would have liked to use that *server hit* to count accesses to the web server. WebMon has a component that monitors every access to a given URL, even though the corresponding request may not come all the way to the web server. With a minor modification, this mechanism also serves for providing an accurate hit count.

In this paper we describe our framework for enabling caching of dynamic objects. We first describe the test Intranet web application in Section 2. We then describe the architecture of the WebMon tool in Section 3. We describe the analysis of the sample application's response time in Section 4. In Section 5 we describe an architecture for converting most of the dynamic objects of the sample application into static objects. We monitored the altered application and report the results in Section 6. Section 7 describes some related work, and we conclude in Section 8 with a summary and guidelines for future work.

2 Corporate Source Web Application

To illustrate our framework, we use the Corporate Source application running within HP Labs [?].

Corporate Source is an Intranet application that enables sharing of source code among HP employees. Figure 1 outlines the Corporate Source application. As shown in the figure, it is a two-tiered web application with a browser accessing web pages using the Apache web server, CGI scripts, and Server Side Include pages.

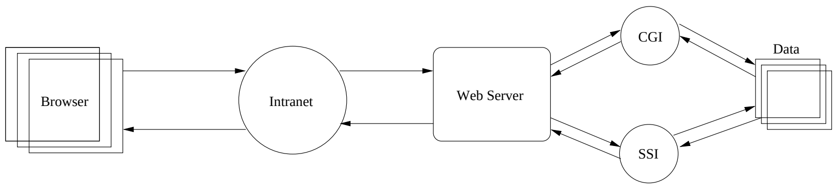

Figure 1: Corporate Source Architecture

Figure 2 shows the main page of the Corporate Source application. Users of Corporate Source conduct the following transactions with the application:

Overview: read an overview of the application.
This is the first page that users see for the application.

Publish: access a form that enables a user to publish new software source code.
This transaction consists of two parts: (a) when the publish form is accessed, and (b) when the publish form is submitted.

Browse: browse existing software source code.
This lists all the projects available in the repository and allows the user to sort the result by either project name, contact person's name, date of submission, or the security classification.

Details: obtain details for an existing software project.
This provides more detailed information about the software projects than the browse item above.

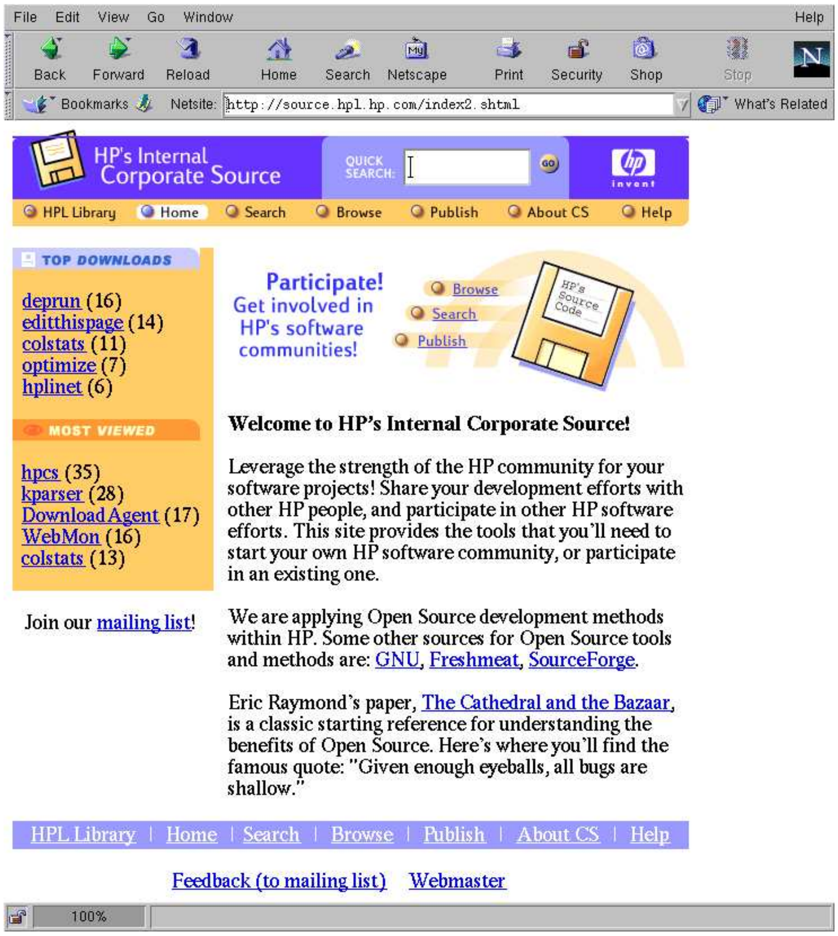

Figure 2: Main page for Corporate Source

`cvsweb`: access the version control system to view the source code history. Access to this information is provided through the `cvsweb` Open Source tool (`http://stud.fh-heilbronn.de/~zeller/cgi/cvsweb.cgi/`).

Search: search for software source code with particular keywords.

About: describes the goals and motivation for the Corporate Source application.

Help: describes how to use the application.

3 Monitoring Architecture

WebMon is a performance monitoring tool for World Wide Web transactions. Typically, a web transaction starts at a Web browser and flows through several *components*: the Internet, a web server, an application server, and finally a database. WebMon monitors and correlates the response times at the browser, web server, and the application server.

WebMon relies on a sensor-collector architecture, as shown in Figure 3. Sensors are typically shared libraries or script components that intercept the actual processing of a transaction request. Each sensor is logically composed of two parts: the *start* part and the *end* part. The start part of a sensor performs data correlation, whereas the end part forwards monitored data to a collector that gathers and further correlates the data. Since the browser is typically behind a firewall, we use a surrogate sensor web server to forward the browser's data to the collector. This web server can be the same as the original web server that is serving the web pages, or a different one. The collector makes the data available in a file, database, or a measurement server, for further processing.

Browser WebMon sensors are built using JavaScript code sent with each instrumented page. The instrumented page instructs the browser to inject the start and end parts of sensors as the event handlers for the appropriate events for the browser.

The start part of the browser sensor generates a request identifier (cf. Figure 3(a)), and passes the identifier along with its request to the web server. The end part of the browser sensor sends its measurement to the WebMon collector (cf. Figure 3(b)).

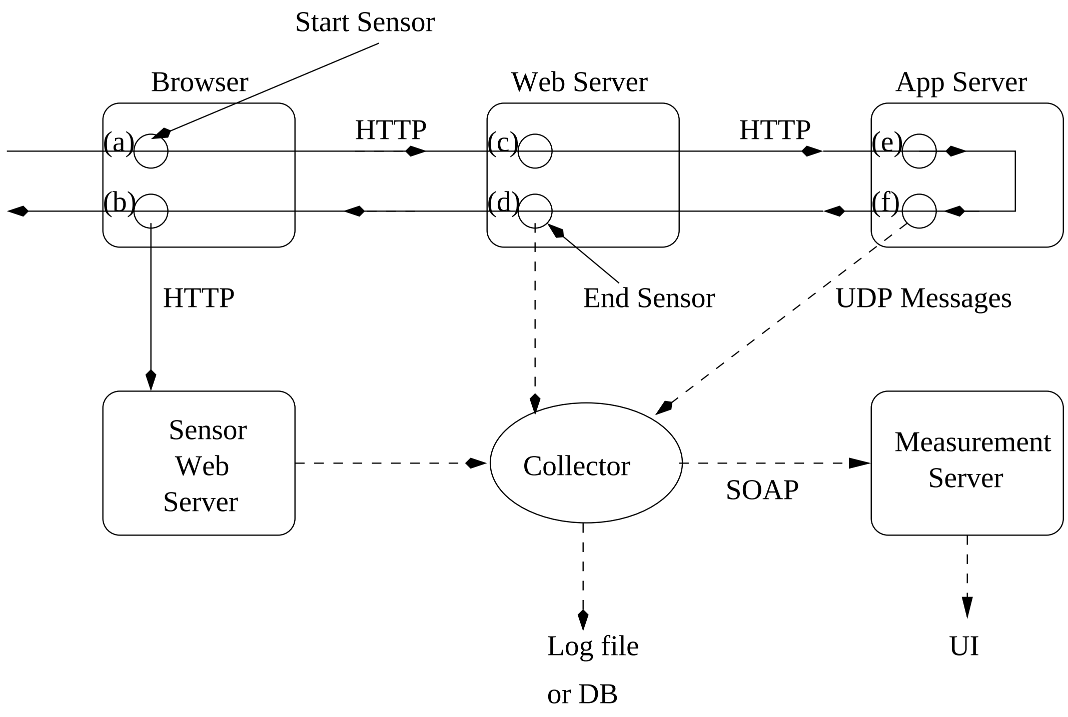

Figure 3: WebMon Architecture

The start part of the web server sensor (cf. Figure 3(c)) extracts the identifier from the request, and passes the identifier along to the application server. The end part of the web server WebMon sensor (cf. Figure 3(d)) sends its measurement along with the request to the collector. Similarly, the application server sensor processes the request and sends its measurement data to the collector (cf. Figure 3(e–f)). The collector correlates the data received by the individual components on the basis of the unique identifier associated with each transaction. In the prototype we use a single collector, although multiple servers tailored for different deployment architectures are feasible.

4 Response Time Analysis

WebMon enables transactional analysis of web applications from a variety of perspectives. In this paper we are mostly interested in the *response time* perceived by the clients for any **transaction** with the application. A transaction *starts* when the user clicks on a URL, or types a URL in the browser window. The transaction *ends* when the user's browser has finished displaying the complete page associated with that URL.

An application usually offers multiple transactions to its users. In addition to an overall transactional view, WebMon enables transactional segregation that can be used to identify poor performing transactions. Similarly, users of applications come from a variety of network *client nodes*. Accordingly, we analyze WebMon data to cluster and identify user locations that are poorly performing. Such views of the data determine opportunities for improving the web application.

As an example, Figure 4 shows the aggregate response times for a two week period for Corporate Source. On the x-axis we plot the hour of the day, whereas on the y-axis (logarithmic scale) we plot the count of client transactions, the server response time (median, min, max; in milliseconds), and the client response times (median, min, max; in milliseconds). The server response times (as the client response times) have been measured using WebMon; a server response time sample corresponds to the difference "$t_d - t_c$" in Figure 3, where t_n is the time stamp made by sensor n; the client response time equals "$t_b - t_a$".

From the figure, we observe that: (1) the client response time samples show a large variance, and (2) the client response times are often very high ($>$

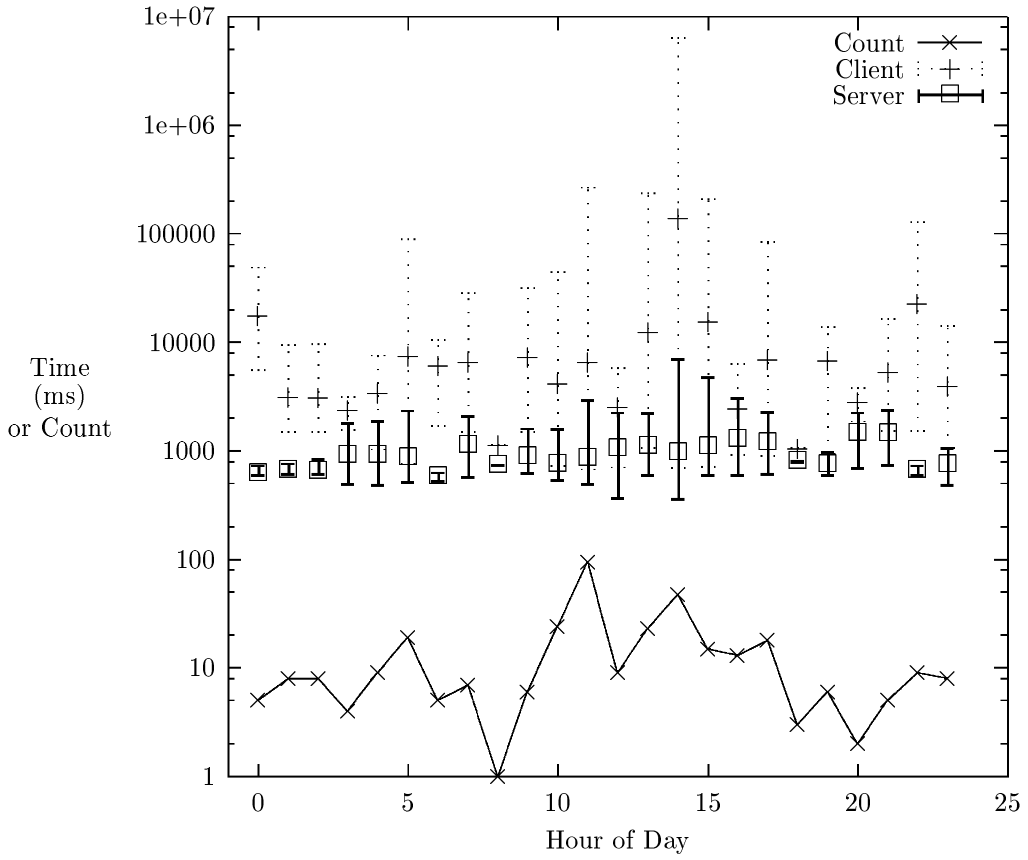

Figure 4: Hourly client and server response times

10 seconds). Both these factors can lead to poor QoE: the users cannot expect a consistent, i.e., highly deterministic, response time from this application nor can they expect a low response time.

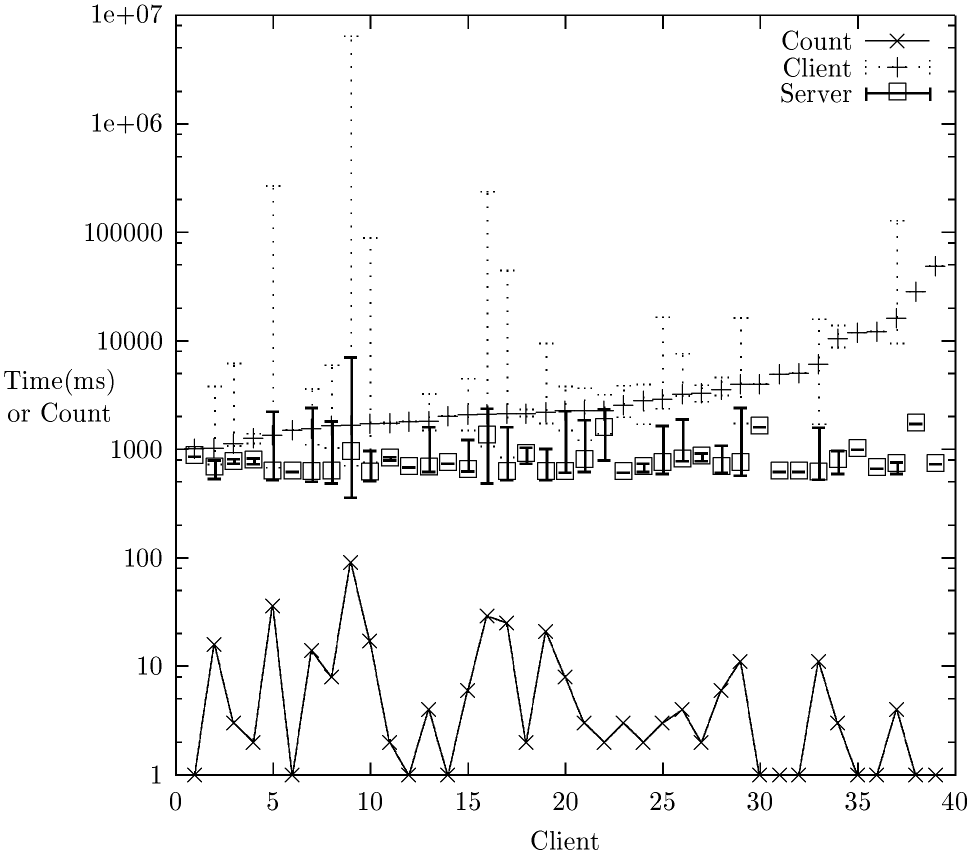

Figure 5: Client and server response times for different clients

Figure 5 shows the same data, however, the the x-axis now represents different request origins for the web service, where the request origin is given by the IP address of the machine that made the request to the web server, hence it could be the proxy between a user's browser and the web server. The data in this figure is sorted by the client's median response time. Note that the client's response time is the one observed by the user's browser and not the requester's response time. Hence, the client time also accounts for any time the request spends between the user's browser and the request origin.

We list some statistics for the client and server response times in Table 1. The clientTime is defined as "$t_b - t_a$" in figure 3; the serverTime is defined as "$t_d - t_c$" in figure 3. The squared coefficient of variation of the sample (denoted as CV^2) is computed as the quotient of the sample variance and the sample squared mean; this measure puts the observed variance in relation to the observed mean[1].

With respect to the clientTime, we observe that its mean value is very high, in fact, much higher than its median. This indicates that there are measurements that are very large, so large, that the clientTime distribution is skewed to the right. This is also confirmed by the extremely high variance and squared coefficient of variation. We attempted to fit a Pareto distribution to these measurements and found that a reasonable fit could only be obtained when the five largest observations are removed, thus implying that these "outliers" have an extreme impact on the observed mean. In the near future we will consider a fit to a suitably chosen hyperexponential distribution using the EM algorithm, cf. [**?**].

For the serverTime, we observe a much better behavior in the sense that the mean, which is much lower, is closer to the median, and the variance is very moderate as well. The squared coefficient of variation is only 0.5, indicating that the serverTime is rather deterministic.

Combining these statistical insights with the fact that the location of the client is an important factor for the perceived performance (cf. Figure 5), we conclude that especially the network, i.e., the Intranet, is responsible for the large variations and large absolute values of the user-perceived performance. The serverTime is always relatively small, and rather deterministic, so it cannot be the cause of the bad performance perceived at the clients.

Remark. At first instance, we were surprised to see the Intranet performance as bottleneck; we had expected the Intranet to show less variable and lower response times. In the case study at hand, however, it seems that the Intranet is behaving more or less like the ordinary Internet. An explanation for this might be the fact that HP is a global company with offices all over the world, and hence, the Intranet is almost as widespread as the Internet.

In order to improve the user-perceived performance, i.e., the QoE, we therefore propose to move the objects requested closer to the clients, in order to avoid, as much as possible, the use of the Intranet. This can be achieved

[1] For comparison: the CV^2 of an exponentially distributed random variable equals 1, of the deterministic variable 0.

	mean (ms)	median (ms)	variance (ms^2)	CV^2
clientTime	24 902.9	1993	1.16413 $\cdot 10^{11}$	187.7
serverTime	917.413	721	3.86337 $\cdot 10^5$	0.46

Table 1: Sample mean, median, variance and squared coefficient of variation of the client and server response times

by increasing the fraction of web-objects that are cached at the clients. Since many of the objects in the study at hand are dynamic objects, we have to devise a means to enable caching of such dynamic objects. This will be discussed in the next section.

5 Enabling Network Caching of Dynamic Objects

To enable caching of some dynamic web objects from Corporate Source application, we required a re-architecture of the application. We describe this re-architecture in Section 5.1. In such re-architectures, determining the appropriate *Cache Update Policies* is quite important. We discuss cache update policies in Section 5.2. Implementation issues are then discussed in Section 5.3, where the issue of counting access is given special treatment in Section 5.4.

5.1 Re-engineering "Corporate Source"

Figure 6 shows how the data for the Corporate Source application is layed out: all data resides in the Unix file system. There are two main data sources: (1) the meta-data source (/usr/local/hpcs), and (2) the source code, with version history (/usr/local/cvsroot).

The Corporate Source application utilizes the two main dynamic page generation capabilities of the Apache web server: (1) server-side includes (.shtml pages), and (2) Common Gateway Interface (CGI) scripts. Out of the nine transactions for the Corporate Source application (presented in Section 2), four are server-side include pages, and five are CGI scripts. Hence, for the entire application, **none** of the pages are cacheable! This fact explains, at least to a certain extent, the measured response times as shown

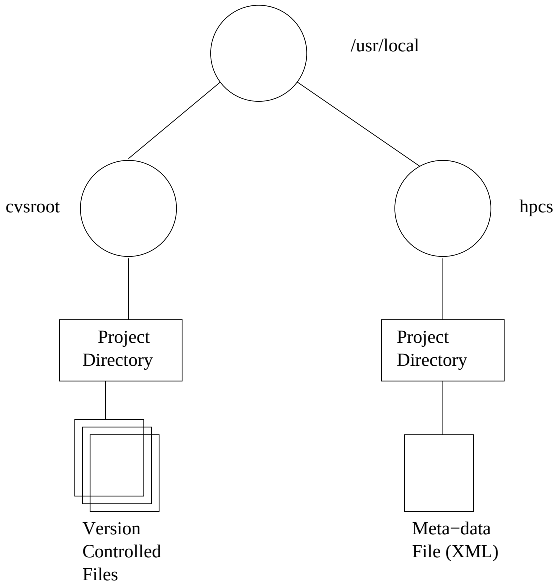

Figure 6: Data layout for Corporate Source

previously: all requests have to be served over the Intranet, over which many other HP run as well.

To improve the responsiveness of this application, we must transform the service from one that is providing non-cacheable web pages to one that is providing cacheable content. We do this by transforming the application architecture, resulting in the architecture presented in Figure 7. The main change in comparison with Figure 1 is that instead of directly supplying the pages from the server-side includes or CGI, the web server supplies a set of HTML pages. The HTML pages, in turn, are periodically updated from the CGI and server-side include pages. Hence, for the web browser, *the web server behaves like a cache rather than a server generating dynamic pages*. Periodically, on requests from the `Cache Generator`, it behaves like a web server generating HTML pages from dynamic web objects. This is similar to the idea of a *reverse-proxy*, as used in several web servers, e.g., see `http://developer.netscape.com/docs/manuals/proxy/adminnt/revpxy.htm`. The effect, however, is quite different, since we *enable the caching of the web pages by intervening network entities*, whereas "reverse-proxy'ing" does not achieve this. Note that for our approach to

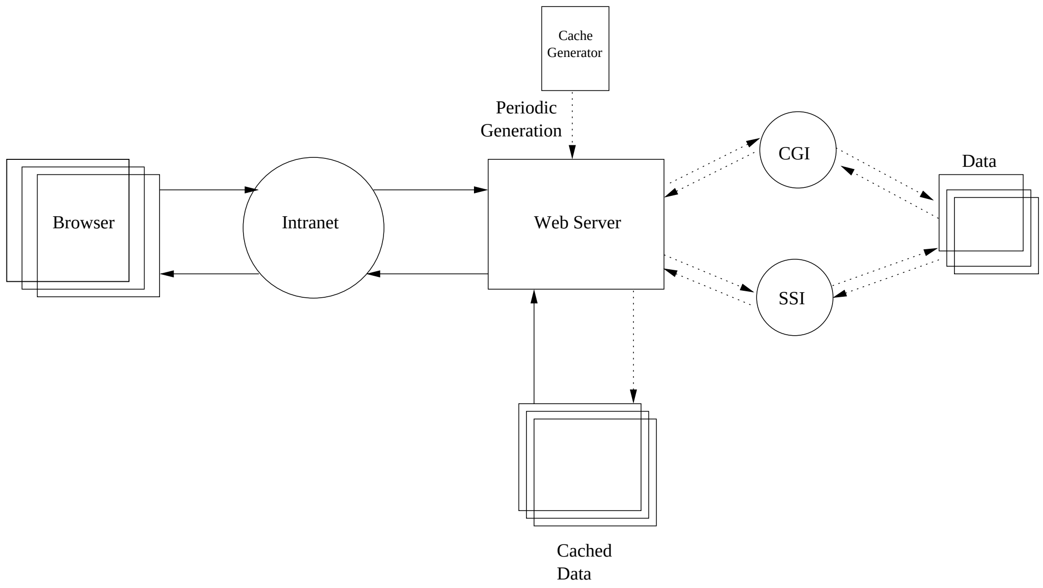

Figure 7: Re-architecture of Corporate Source

operate correctly, all original HTML links to dynamic objects have to be converted to links to their static counterparts.

5.2 Cache Update Policies

The re-engineering process involves creating a cache of the dynamically generated web pages, and updating these cached pages as needed. For this purpose, it is useful to classify web pages as follows:

- **on-demand** pages are generated on demand; they are dynamic pages;
- **on-change** pages are generated when any underlying content for the page changes;
- **periodic** pages are generated periodically.

In the case of on-demand pages, the information given to the user is always up-to-date, but the disadvantage is possibly poor response time for the end-user. For on-change pages, the users may obtain up-to-date pages, but the machinery required to monitor all possible changes is complex and may be expensive for the web server, e.g., see Challenger et al. [**?**]. For pages that are updated on a periodic basis, the users may sometimes receive stale information, but the machinery and effort required by the web server are not expensive. Hence, we decided to implement the periodic update policy.

To see how good the periodic update policy is, we consider the following analytical model. Define: the object request rate (λ), the object change rate (τ), and the cached object update rate (γ). We *assume* that both the process of object requests and the process of object changes are Poisson processes. Furthermore, due to the periodic update policy with rate γ, the inter-update time is a constant, denoted $D = 1/\gamma$.

We can now view the time-line as partitioned by updates, all exactly D time units apart. Consider the "arrival" of an object request (as part of a Poisson process). The time that has passed since the last periodic update is a random variable Y, with distribution

$$F_Y(y) = \frac{1 - F_X(y)}{E[X]},$$

with F_X the inter-update time distribution and $E[X] = D$ the mean inter-update time [**?**, page 102]. $F_X(x)$ is a unit step-function at $x = D$. Accordingly, Y is uniformly distributed on $[0, D]$. This is intuitively appealing; the page change occurs truly randomly in an update interval.

Similarly, we can compute the distribution of Z, the time until the most recent page change. Since the page changes form a Poisson process as well, the time between page changes is a negative exponentially distributed random variable, which is, by definition, memoryless. Hence, Z has a negative exponential distribution with rate τ as well.

The probability that the request for the page will result in an up-to-date copy, then equals the probability $\Pr\{Y < Z\}$, that is, the probability that the last page update took place more recently than the last page change. Conditioning on the time Y until the last update instance, we obtain:

Conditioning on the time s until the last update instance, we obtain:

$$\begin{aligned}\Pr\{\text{"okay"}\} &= \Pr\{Y < Z\} = \int_0^D \Pr\{Y < Z | Y = t\} \Pr\{Y = t\}\, dt \\ &= \int_0^D \frac{e^{-\tau t}}{D} dt = \frac{1}{\tau D}\left(1 - e^{-\tau D}\right). \qquad (1)\end{aligned}$$

One can easily show the following properties for this probability, which we for convenience write as $p(x)$, with $x = \tau D \geq 0$. The following limits are obeyed by $p(x)$: $\lim_{x \to 0} p(x) = 1$ and $\lim_{x \to \infty} p(x) = 0$. Furthermore, the derivative $p'(x) < 0$ for $x > 0$, so that $p(x) \in [0, 1]$, and it decreases monotonously from 1 to 0, for $x \geq 0$.

As an example, consider the case where the update rate $\gamma = 10$ per month (every third day an update; $D = 0.1$) and the change rate $\tau = 1$ (per month). The probability that *a* request will result in the right (most recent) copy of the page then equals $p(0.1) = 10(1 - e^{-0.1}) = 0.95$. If we set $\gamma = 30$ (daily updates; $D = \frac{1}{30}$) we obtain success probability 0.984. Notice that the current model does not have λ as one of its parameters. The probability $p(x)$ holds for ever request that arrives as part of a Poisson process.

5.3 Implementation considerations

To implement the periodic updates, we re-engineered the web application to use static HTML pages instead of server-side includes and CGI scripts. We first created a sub-directory "Cache" that contains all the static pages offered by the web server. A Unix "cron" job periodically accesses the CGI and server-side include web pages and generates the static pages for the web server. In the following, we use an example to illustrate the method of converting dynamic pages to static pages.

As described in section 2, one of the transactions provided by the Corporate Source application is to describe the details of a given software project, using the meta-data stored about the project. This meta-data is stored as an XML file under the directory:

```
META-DATA/<project-id>/<project-id>.xml
```

To show the meta-data, a CGI PERL script parses the XML file and generates the HTML code for the user's browser to display. The <project-id> is a parameter to the script, through the fat URL method. Hence, the URL to ask information for the project "hpcs" would be:

```
http://src.hpl.hp.com/cgi-bin/sdata.cgi?software=hpcs
```

To develop an effective cache for such web pages, we need a script that converts the XML files into the required HTML files. This script should run only when data in the META-DATA directory has changed. Also, this script should run for all projects submitted to the corporate source, even projects that may not have been submitted when the script was first written, i.e., we cannot hard-code the projects in the script.

An elegant solution for such a script can be developed using the Unix `make` utility [?]. `make` is a utility that takes a specification (usually called `makefile`) of dependencies among source and derived files, and when executed with the right specification (re-)produces all derived files for which the sources have changed. The user specifies the tools used for transforming the source to the derived files. For our case, we know the source files as the XML files that contain the meta-data. The derived files are the HTML files to be generated. We can use the existing web server and its CGI script as the tools to transform the XML to HTML.

Fortunately, `make` provides some rudimentary control concepts in its specification language for us to write this transformation efficiently. The `Makefile` for Corporate Source service is about a hundred lines of code. Below is a part of it that performs the transformation for the software details page mentioned above:

```
 1 url = http://localhost
 2 sdatadirs = $(wildcard $(METADATA)/*)
 3 sdata : $(INCLUDES) $(BRWSCONTENTS) $(MONITOR) $(METADATA)
 4        $(foreach dir,$(sdatadirs), \
 5         $(shell perl geturl.pl $(url)/browseA.shtml > sdata$(notdir $(dir)).htm) \
 6         $(shell perl geturl.pl $(url)/cgi-bin/sdata.cgi?software=$(notdir $(dir)) \
 7           >> sdata$(notdir $(dir)).htm) \
 8        $(shell perl geturl.pl $(url)/browseB.shtml \
 9           >> sdata$(notdir $(dir)).htm))
10        touch sdata
```

In this transformation rule, line 2 identifies all the software projects currently in the Corporate Source using the `wildcard` feature of Make. Line 3 creates a dummy target `sdata` that will be used to determine if any of the meta-data files changed or not. Lines 4–9 loop through each of the software project's XML files and generate the corresponding HTML files by making calls to the web server. The calls to the web server are accomplished using a PERL script "geturl.pl", which depends on the PERL HTTP USER-AGENT module.

Note that the above transformation does not create the cacheable objects independently, but in a batch. For the current Corporate Source application, with about a dozen projects, this is sufficient. For a larger application this could become an issue, in which case a more efficient Make specification must be written that checks for changes in individual projects and (re-)generates only the HTML pages for the changed projects.

5.4 Counting Accesses

As we will show in the next section, the redesign of the Corporate Source application does significantly improve the user-perceived performance. It does, however, create one problem: in the new architecture, some of the user requests may not make it anymore to the actual web server, so that we do not know anymore how many users have viewed a particular software project, or how many have downloaded it. With the dynamic pages it was easy to obtain this information from the web server logs. This seems to be a general problem when a dynamic page is changed to a static, cacheable page.

To overcome this problem, we again used the WebMon tool [?]. Recall from Section 3 that in order to send browser response times back to a measurement data collector, WebMon introduced an extra HTTP request back to a web server that forwards the measured data to the measurement collector. We noted that this additional web server could be another web server, on a different network, in case performance of the original web server was an issue. In our case, this additional WebMon request can be used to log accesses to the server, and hence give us the count of dynamic web page access.

To achieve this, we must modify the original WebMon client side instrumentation to include the URL of the previous request in the data collection request. The modified JavaScript code is presented in an appendix (with the only modification in comparison to [?] on line 45).

6 Analysis of the data with cached pages

We adapted the Corporate Source application as described above, and subsequently monitored it for about two weeks. The measurements for client and server time and count are presented in Figure 8, ordered according to client origin.

From the new data set of 521 observations, we computed the sample mean, median, variance and squared coefficient of variation, as given in Table 2. Note that the serverTime could not be monitored for the cached pages, simply because request to these pages do not make it to the server. The serverTime has been observed for only 166 of the 521 entries of the data set.

The serverTime has an extremely small coefficient of variation, which means that there is hardly any variability observed. An Erlang-k distribution with $k \approx 5$ could be used to describe the data. We also observe that the

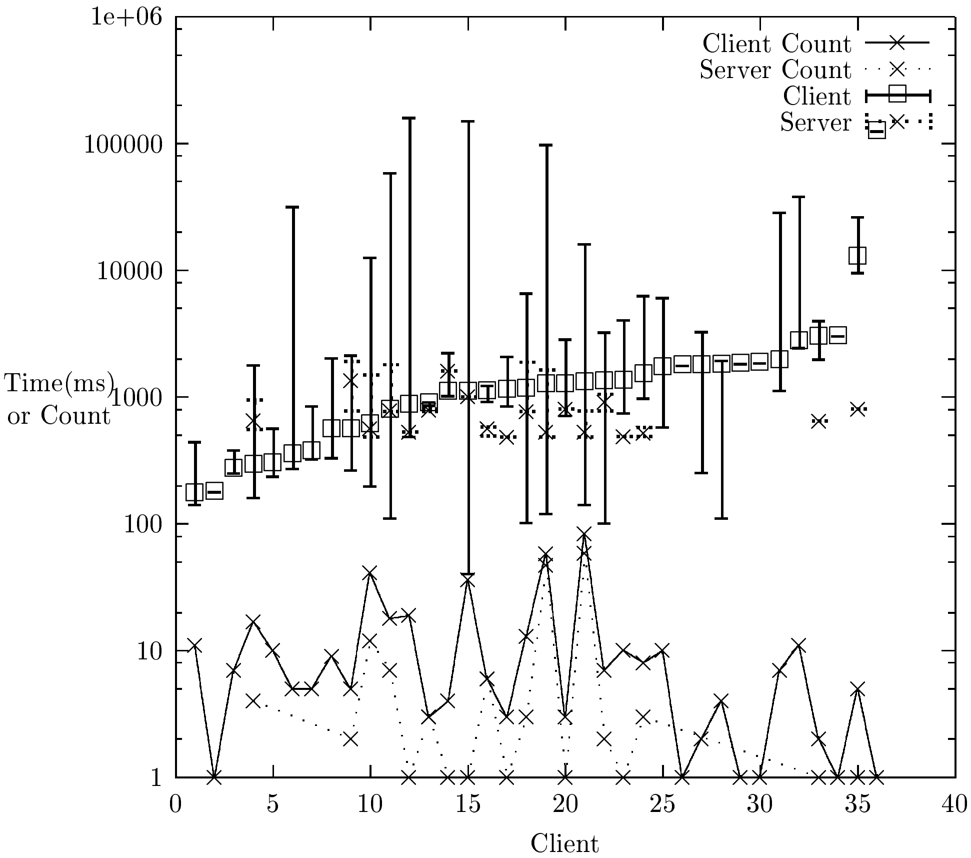

Figure 8: Client and server count and response times for different clients

serverTime has decreased (the mean serverTime has decreased to 69% of the former value, the median to 74%), most probably due to the decreased load on the server.

	mean (ms)	median (ms)	variance (ms^2)	CV^2
client	4524.3486	1121	$5.209 \cdot 10^8$	25.4
client-15	1081.2	1549.061	$5.3033 \cdot 10^6$	2.21
server	638.988	539	81219.2	0.2

Table 2: New sample mean, median, variance and squared coefficient of variation for serverTime and clientTime

The mean clientTime is reduced to only 20% of its former value, however, the variance of the clientTime is reduced by three orders of magnitude (which corresponds to a reduction of roughly 99.5%). The coefficient of variation is still large, but considerably smaller than before. The same is true for the variance. Also observe that the mean-to-median ratio (previously equal to 12.5) has reduced to only 4.0; the clientTime distribution is considerably less skewed now.

Interestingly, if we drop the 15 largest observations for the clientTime, the variance is reduced by another two orders of magnitude and end up with a "normal" coefficient of variation. This shows that a few extremely long response times cause these unusual statistics; the majority of the observations is much more moderate. More advanced statistical methods are required to filter these outliers.

7 Related Work

Rajamony and Elnozahy [**?**] describe an approach for client-side response time monitoring similar to WebMon. Unlike WebMon, however, their tool does not correlate the client-side data with the server-side data. Hence, even if their data shows poor web application performance, it will not enable attribution of poor performance to the network or the server.

Challenger et al. [**?**] present a scheme for maintaining in-memory cache for dynamically generated web pages. They use a *object dependency graph* (ODG) to determine data pages that may be updated when the underlying data changes. Although conceptually similar to our approach, their caching scheme has significant differences from the work presented in this paper: (1) we store the cached pages as HTML pages on disk, hence our scheme is resilient to server crashes, (2) the structuring of our data as files allows us to take advantage of existing dependency tracking tools like `make` [?] rather than build our own dependency tracking machinery, (3) the URL's in our scheme point to HTML pages rather than to dynamic pages. This way we can leverage the variety of intervening caches between a web client and server. Challenger et al., however, can have much more fine-grained control on their update propagation using a weighting scheme for object dependencies. Finally, we use a periodic update scheme to update cached pages, as our web pages do not change that often, whereas they use a trigger-based update scheme.

SpiderCache (`www.spidercache.com`) is also an in-memory caching technique for dynamic web objects. When a dynamic web object is first requested, SpiderCache keeps a copy in memory such that subsequent requests for the same object can be served from the memory instead of performing the queries and computations necessary to derive the HTML page. SpiderCache implements several heuristics to determine whether it can serve a page from memory or whether it has to request a newly generated page from the application server. SpiderCache, however, does not have any way of enabling network caching of dynamic objects as reported in this paper. The request always has to first come to the SpiderCache server before it can be answered.

FineGround Networks (`www.fineground.com`) uses a somewhat different idea for speeding up access to dynamic web objects. They provide a web object **condensor** that is able to communicate with a user's web browser on subsequent requests to the same web object. On subsequent requests, instead of sending the entire web page, the condensor is able to send just the modifications since the last request. This way, FineGround can save bandwidth for both the clients and the servers. The reduction in response time, however, may not be as much due to the communication latency between the browser and the server. Also, the condensor works on a different class of dynamic web objects than the ones discussed here: it works on dynamic objects that change frequently for the same user, whereas our approach works for dynamic objects that change infrequently.

8 Conclusions

In this paper, we have proposed and evaluated a performance enhancement of an Intranet web application, resulting, for the case study at hand, in a mean response time reduction of 80% and in a response time variance reduction of three orders of magnitude.

The reengineering of the web application was motivated by the insights gained with the novel tool, WebMon, that can monitor end-user perceived response times for web applications and correlate them with server response times. WebMon helped us in determining the performance bottleneck at hand, the is, the Intranet. In the subsequent reengineering process of the web application studied, we took measures to reduce the load on the Intranet. In particular, we proposed a method which transforms dynamically generated web pages (that cannot be cached) into static web pages. Thus, this transformation enables caching of web objects at the client, and therefore reduced network traffic and load.

Our approach relies only on software tools that are widely available (like the unix `make` facility), so that it can be used by other web application developers for improving the responsiveness of their web applications.

Although useful for a large number of web applications with dynamic objects, our approach is not directly suitable for web applications with frequently changing objects, such as commonly found in personalized e-commerce shopping web applications and news suppliers. In the future, we will address the performance issues for such web applications as well.

A Adapted WebMon JavaScript code

```
<SCRIPT language="javascript">
var wm_netscape=navigator.appName.indexOf("Netscape");

function wm_get_cookie(name) {
  var start=document.cookie.indexOf(name+"=");
  var len=start+name.length+1;
  if (start==-1) return "0";
  var end=document.cookie.indexOf(";",len);
  if (end==-1) end=document.cookie.length;
  return unescape(document.cookie.substring(len,end));
}
```

```

function wm_sensor_start(event) {
  var ms=new Date().getTime();
  clickCookie=escape(ms+"-"+document.URL)+";PATH=/";
  document.cookie="click="+clickCookie;
  return true;
}

function wm_checkReferrer(clickCookie){
  if (clickCookie=="0") return false;
  dash=clickCookie.indexOf("-");
  if (dash==-1) return false;
  creferrer=clickCookie.substring(dash+1);
  if ((creferrer!=document.referrer) && (creferrer!=document.URL))
    return false;
  else
    return true;
}

function wm_getID(clickCookie){
  if (clickCookie=="0") return "0";
  dash=clickCookie.indexOf("-");
  if (dash==-1) return "0";
  return clickCookie.substring(0,dash);
}

function wm_sensor_end(event) {
  clickCookie=wm_get_cookie("click");

  if (wm_checkReferrer(clickCookie)){
   var ms=new Date().getTime();
   document.cookie="load="+wm_getID(clickCookie)+"x"+ms+";PATH=/";
   myImage = new Image() ;
   myImage.src = "/webmon.wmi" + ‘‘?’’ + document.URL;
  }
  return true;
}
</SCRIPT>
```