SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2004; **34**:1381–1395
Published online 23 August 2004 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.617

Enforcing direct communications between clients and Web servers to improve proxy performance and security

Songqing Chen[1] and Xiaodong Zhang[2,*,†]

[1]*Department of Computer Science, George Mason University, Fairfax, VA 22030, U.S.A.*
[2]*Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795, U.S.A.*

SUMMARY

The amount of dynamic Web contents and secured e-commerce transactions has been dramatically increasing on the Internet, where proxy servers between clients and Web servers are commonly used for the purpose of sharing commonly accessed data and reducing Internet traffic. A significant and unnecessary Web access delay is caused by the overhead in proxy servers to process two types of accesses, namely dynamic Web contents and secured transactions, not only increasing response time, but also raising some security concerns. Conducting experiments on Squid proxy 2.3STABLE4, we have quantified the unnecessary processing overhead to show its significant impact on increased client access response times. We have also analyzed the technical difficulties in eliminating or reducing the processing overhead and the security loopholes based on the existing proxy structure. In order to address these performance and security concerns, we propose a simple but effective technique from the client side that adds a detector interfacing with a browser. With this detector, a standard browser, such as the Netscape/Mozilla, will have simple detective and scheduling functions, called a *detective browser*. Upon an Internet request from a user, the detective browser can immediately determine whether the requested content is dynamic or secured. If so, the browser will bypass the proxy and forward the request directly to the Web server; otherwise, the request will be processed through the proxy. We implemented a detective browser prototype in Mozilla version 0.9.7, and tested its functionality and effectiveness. Since we have simply moved the necessary detective functions from a proxy server to a browser, the detective browser introduces little overhead to Internet accessing, and our software can be patched to existing browsers easily. Copyright © 2004 John Wiley & Sons, Ltd.

KEY WORDS: browser; dynamic contents; proxy caching; secured transactions

*Correspondence to: Xiaodong Zhang, Department of Computer Science, College of William and Mary, Williamsburg, VA 23187-8795, U.S.A.
†E-mail: zhang@cs.wm.edu

Contract/grant sponsor: National Science Foundation; contract/grant numbers: CNS-0098055 and CCF-0129883

Received 30 September 2002
Revised 10 March 2004
Accepted 22 March 2004

1. INTRODUCTION

Proxy servers were originally designed for caching static Web contents that are files stored in a Web server, and have been effectively used for this purpose[‡]. Proxy servers also have to deal with other ever emerging types of Web contents. In this study, we examine the proxy's roles in processing dynamic Web contents and secured transactions, and present a software method to improve the Web access performance and security.

Dynamic Web contents are generated by programs executed at the requesting time. Although the response time to access a dynamic Web page is several orders of magnitude slower than that to access a static page, the amount of dynamic Web content services in commercial, government, and industrial applications has been dramatically increasing. Researchers have examined the percentage of dynamic contents in several highly popular Web sites including the Melissavirus site, the eBay site, the 1998 Olympic Winter Game site, the Alexandria Digital Library site and others, and found the percentage ranges from 10–42% [1–3].

A number of methods had been devised to improve the performance of dynamic Web content services based on the current Web access infrastructure, focusing on effectively caching and processing dynamic Web pages. One representative approach from the server side is to cache dynamic contents in the Web servers or in dedicated storage close to the servers [1,3–6]. The approach from the proxy side is to restructure existing client-side proxy servers to be capable of some Web server processing and caching functions for dynamic contents [2,7–9]. Many studies have shown that caching dynamic contents at the server side is most effective and more appropriate [1,3,5,6,10]. In other words, dynamic contents are continuously changing and not suitable for client-side proxy caching, and thus it is not beneficial for a proxy to keep them. Although most proxy servers do not cache dynamic Web contents, a proxy has to make connections to Web servers and temporarily place a document while its dynamic nature is detected for the purpose of a replacement or deletion.

Internet e-commerce services have become popular and are provided by many company servers. Furthermore, an ever-increasing amount of business transactions are completed on-line. E-commerce requires a secure channel to complete these transactions. Since the standard HTTP protocol is not sufficiently secure, the SSL was proposed [11], and is commonly used for secure data transmission, such as on-line shopping and on-line credit card payment. Since the secure transactions must not be intercepted or cached by any intermediary, a proxy has to tunnel the communication between the client and the server when such contents reach the proxy. The involvement of the proxy can be a serious security concern besides the unnecessary processing overhead.

Instead of further investigating Web server caching or enhancing the proxy caching ability for dynamic contents and secured transactions, we have focused our efforts on eliminating the client-side proxy processing overheads, and providing a reliable way for secure transactions. This technique is also complementary to the server-side caching approach, further reducing response times to clients and removing unnecessary processing burdens on the proxy and unnecessary risks for secured transactions. In the rest of the paper, the term 'proxy' or 'proxy server' means a client-side proxy if not specifically noted.

[‡]Proxy servers can also be used as firewalls for security reasons.

In this study, we will first show that this ignored proxy processing overhead is significant. We also look into the security risks of tunneling secured transactions. Conducting experiments on the proxy Squid2.3-STABLE4 [12], we have quantified this unnecessary processing overhead to show its significant impact on increased client access response times. We show that the average additional time spent on the Squid proxy to process a dynamic document is about 10–30% of the average response time for directly accessing the Web server with the caching ability for a dynamic document, and is 3–10 times higher than that for accessing a static document in the Web server.

The performance results have led us to consider restructuring the organization of representative proxy systems, aiming to reduce or eliminate the processing overhead and the unnecessary risks in the proxy. For the dynamic contents, we discuss several possible techniques, and present technical difficulties that prevent us from achieving our goal. We conclude that it may not be possible to find an effective way to solve the overhead and the security problem by restructuring proxy servers.

In order to eliminate the overhead portion of the response time to accessing dynamic Web contents and the unnecessary risks for secure transactions, we propose a simple but effective technique that enhances a standard browser, such as the Netscape/Mozilla, to be able to detect and schedule the outgoing requests, which we call the *detective browser*.

Upon an Internet request from the user, the detective browser can immediately determine whether the requested content is dynamic, or it requires a secured channel. If so, the browser will bypass the proxy and send the request directly to the Web server instead of going through the proxy. Otherwise, the request will be sent to the proxy as usual. We have implemented a detective browser prototype and tested its functionality and effectiveness on a text-based browser. We have also implemented it in the Mozilla.0.9.7.

Besides the response time improvement, our study also cares about the improvement of system reliability and availability. Bypassing a large portion of unnecessary dynamic contents and secure transactions from proxy caches, we are able to effectively reduce the workload of a proxy to improve its reliability. For a commercial proxy, our approach will also improve its cost-effectiveness.

Since the necessary detection function is simply moved from a proxy to a browser, and the detection can be done by scanning the URL only once, the detective browser introduces little overhead to Internet accessing, and our implementation can easily be patched to any existing browser to provide an additional option for users. An inevitable trend in Internet computing is to widely decentralize the usage of distributed resources. Such an example is the peer-to-peer computing model, where a peer is both a server and client. Although the detective browser cannot play a server's role, it can facilitate a server's duty of content type detections. Similar effort has been made to add some intelligence to clients for the purpose of scalability [13].

The paper is organized as follows. In the next section, we first overview proxy structure and how dynamic contents are temporarily sheltered in the proxy, then we discuss the technical difficulties in eliminating or reducing the processing overhead in the proxy. Proxy tunneling is analyzed in Section 3. In Section 4, we present our evaluation methodology and measurement results to quantify the average processing overhead time in the proxy. We present the design and implementation of the detective browser in Section 5 and its performance analysis in Section 6. We conclude the paper in Section 7.

2. SHELTERING DYNAMIC CONTENTS IN PROXY AND THE OVERHEAD

Before we discuss how a proxy processes dynamic contents, we briefly overview its basic procedure of processing requests. Upon a Web page delivery request, the proxy first checks if the page is available and valid. If so, the proxy will deliver the page to the requesting client. If the page is available but it is not valid, the proxy will send an IMS (If-Modified-Since) message to the server to check whether the contents have been changed. The server will either send an updated page to the proxy or inform the proxy that the page has not been changed. The proxy will then either send the updated page or the original page to the requesting client. If the requested page is not cached in the proxy, then the request will be forwarded to the server. Upon receiving a reply from the server, the proxy will store the page in the local memory/disk, as well as forward a copy of the page to the requesting client. As soon as the header of the page is received from the server, the proxy is able to decide if the page is cacheable or uncacheable[§]. The replacement policy will work to reclaim the space by deleting LRU or unusable pages at a certain frequency.

2.1. How are dynamic contents processed in the proxy?

A representative proxy is the Squid proxy. It uses the same procedure to process requests for dynamic contents that are uncacheable in the proxy as that used for static contents. The time and space used to process the dynamic contents are true overhead because the contents will not be reused by other clients. The sequence of steps followed in proxy squid2.3-STABLE4 to process requests for dynamic contents is as follows.

- Upon receiving a request from the client (using function *clientReadRequest()*), the proxy parses the request and processes the headers (using functions *parseHttpRequest() and urlParse()*, respectively). The access right of the request will be checked (using function *clientAccessCheck()*) after the redirection is done (using function *clientRedirectDone()*). Then the proxy will process the request (using function *clientProcessRequest()*) by filling in some known attributes in the data structure, and determining if the requested content is in the proxy. Since the request is for a dynamic content, it has not been cached in the proxy.
- The proxy forwards the request to the Web server (using functions *clientProcessMiss()* and *fwdStart()*) after finding no peer proxy (using function *peerSelect()*). A TCP connection will be started (using function *fwdConnectStart()*) if a persistent connection is not used, via which the request will be sent to the server (using function *httpSendComplete()*).
- When the server returns the generated dynamic content, the proxy allocates a block of memory to store the content (using function *httpReadReply()*). The proxy detects that the content is uncacheable after parsing the header (using function *httpProcessReplyHeader()*). The proxy makes the dynamic content private (using function *httpMakePrivate()*). (In contrast, if the content is static, the proxy will use *httpMakePublic()* to make the file public.) Even if the

[§]The dynamic or static nature of the Web contents is determined by the Web server. Whether the content is cacheable or uncacheable may be suggested by the Web server by setting the reply headers or by the request by setting request headers, and determined by the proxy. Not all static contents are cacheable, but most dynamic contents are uncacheable.

dynamic content expires and will not be reused, the proxy has to buffer the content temporarily in the memory so that the data can be copied and sent to the client, by using the function *storeClientCopy2()*).
- Since the stored dynamic content is not usable again, it will get released and flushed out of the memory.

In each step, corresponding data structures will be created and allocated for processing. Related operations for these data structures are nested. The space and processing time involved in these operations delay the response time to the clients accessing dynamic contents.

2.2. Technical difficulties in eliminating the overhead

'Can we eliminate or reduce the overhead by detecting the dynamic content as early as possible in the proxy?' Having raised this question, we have tried to provide solutions for it. The dynamic nature of a request can be detected if the proxy further parses the request immediately after the request is received.

The implementation of this early detection in the proxy is straightforward with little overhead. With this early detection ability, a proxy has the following three alternatives to deal with a dynamic content request.

- *Making the Web server contact the client directly.* After detecting the dynamic nature of a request, the proxy asks the Web server receiving the request for the dynamic content to contact the client directly, instead of sending the document to the proxy. The proxy processing overhead will be eliminated because the proxy will never receive dynamic contents from Web servers. Unfortunately, this proposal is not practically useful although it is technically possible. In current Internet infrastructure, the data communications for a request from a client and its reply from the proxy are fixed in a pair of ports. In order to make the server contact the client directly, each client must be capable of listening to multiple connections because the reply for a request may come from a site that is different from the targeted destination. In addition, the socket used by the client to send the request needs to be terminated if the reply does not come from the proxy. A new connection between the client and the Web server must be created after that. The additional cost and existing Internet infrastructure make it impossible for this proposal to be implemented.
- *Making the client contact the Web server directly.* After detecting the dynamic nature of a request, the proxy declines a dynamic content request by sending a message back to the client to ask it to contact the Web server directly. Thus, the proxy processing overhead will be eliminated. However, the overhead times spent on the client request and the proxy declination will increase the response time of accessing dynamic contents. (Our measurements indicate the overhead is up to 20% of the response time.)
- *Making the proxy not shelter the dynamic contents.* After detecting the dynamic nature of a request, the proxy processes the request as the existing proxy does. However, the received dynamic content will not be cached. In other words, the content will be flushed out of the memory soon after it is forwarded to the client. This approach can certainly save the proxy space, but the processing time overhead and proxy load burden remain, because the space maintenance (using the function *storeMaintainSwapSpace()* in Squid) is overlapped with the proxy operations on the clients' requests and servers' replies. This is the best that the current proxy can do.

Discussing several alternatives based on existing proxy structures, we have presented the technical difficulties in eliminating the processing overhead even if the proxy detects the dynamic nature of a client request in the earliest stage.

3. TUNNELING HTTP COMMUNICATIONS BETWEEN CLIENTS AND SERVERS THROUGH PROXY

Before we look into the details about how the proxy tunnels the HTTP communications, we give a brief overview of the SSL, which is atop of TCP/IP for the secured data transmission. SSL is an open, non-proprietary protocol proposed by Netscape Inc. [11], which has become the most common way to provide encrypted data transmission between Web browsers and Web servers on the Internet. Built upon private key encryption technology, SSL provides data encryption, server authentication, message integrity, and client authentication for any TCP/IP connection. Most commercial Web sites provide secured services to the clients based on the SSL.

To tunnel the communication between the client and the server, a CONNECT method is used (instead of the normal GET). The CONNECT method is a way to notify the proxy to tunnel the incoming contents. The SSL session is established between the client who sends the request, and the Web server who generates the reply; the proxy between the two parties merely tunnels the encrypted data, simply passing bytes back and forth between the client and the server without knowing the meaning of the content.

3.1. How is the tunneling done in the proxy?

In Squid 2.3, the tunneling is done for both the client and the server as follows upon an SSL session request.

- Upon receiving a request for secured transactions from the client by function *clientReadRequest()*, the proxy parses the request and processes the headers by functions *parseHttpRequest()* and *urlParse()*. The access right of the request will be checked by function *clientAccessCheck()*, after the redirection is done by using the function *clientRedirectDone()*. Then the proxy will process the request using the function *clientProcessRequest()*, in which the CONNECT method will be identified and function sslStart() will be called to start the tunneling.
- The proxy uses the function *sslReadClient()* to read the client request and queues it for writing to the server. Functions *sslSetSelect()* and *sslWriteServer()* are then called to write data from the client buffer to the server side.
- When the proxy gets a reply from the Web server, it calls the function *sslReadServer()* to read from the server and queue it for writing to the client. Functions *sslSetSelect()* and *sslWriteClient()* are then called to write data from the server buffer to the client side.

In Squid 2.5, the program becomes more complicated, since Squid can encrypt or decrypt the connections with the help of the OpenSSL [14]. However, the tunneling principle is the same.

3.2. The potential security problems of the proxy tunneling function

Besides the additional proxy overhead to tunnel the communications between the client and the server (we will present our measurement results later in the paper), tunneling can be a potential source of security problems [15]. The main concern is that a proxy can be used to relay bogus transactions. This can be done as the number of ports used for communications is generally limited. The port used for tunneling can be the target of attacks even if it is not a reserved one. An example of an attack is a HTTP client who on connecting to port 25 could relay spam via SMTP. As reported by the IRCache group [16], the IRCache proxies have been used anonymously to make fraudulent credit card purchases. They decided to deny all SSL requests. Thus, we strongly argue that the proxy should not be involved in any secured transaction.

4. ANALYSIS AND MEASUREMENT OF THE OVERHEAD FOR SHELTERING AND TUNNELING

As discussed in Section 2.1, a dynamic document request will be processed by a sequence of four steps in the proxy—the same as used for static contents requested for the first time, although the obtained dynamic document will not be used. In each step, processing time and/or storage space are consumed. In step 1, the overhead operations involved are receiving, parsing, checking, redirecting the request, and a final request miss in the proxy. In step 2, the proxy will make a peer selection and a socket connection to the target server. The major delay in this step comes from a TCP connection and slow start. In step 3, the overhead operations involved are receiving, reading, parsing and storing the requested dynamic document. A memory block must be allocated to receive the document, and disk space is allocated to store the document. Finally, in step 4, a replacement operation is used to delete the obtained document.

The time and space involved for the above operations are true processing overhead. In this section, we will quantify the overhead by measurement.

4.1. Processing overhead measurement for Sheltering

Figure 1 presents a basic measurement structure of the processing overhead for dynamic contents in the proxy. There are two sets of measurements: (1) the average response time from a client to directly request a set of dynamic documents (represented by the solid arrows in Figure 1), and (2) the average response time from the same client to request the same set of dynamic documents through a proxy (represented by the dotted arrows in Figure 1). The difference between (2) and (1) is the ideal average processing overhead in the proxy. The Squid proxy is used in our experiments. The ‘client’ we used in the experiments was not a normal browser, but a text-based program. Its functions are to send out the requests, and wait for the reply from a Web server or from a proxy server. After the requested document is received, the client completes its job. It does not involve the normal browser functions of displays and other services (defined in Netscape/Mozilla), since they may cause more unstable factors.

Intuitively, the overhead measurement should involve necessary instrumentation in the proxy and the use of workloads of dynamic Web contents. We did the instrumentation, but found that the results were very unstable with a high intrusive effect. Examining the experiments, we realized that there

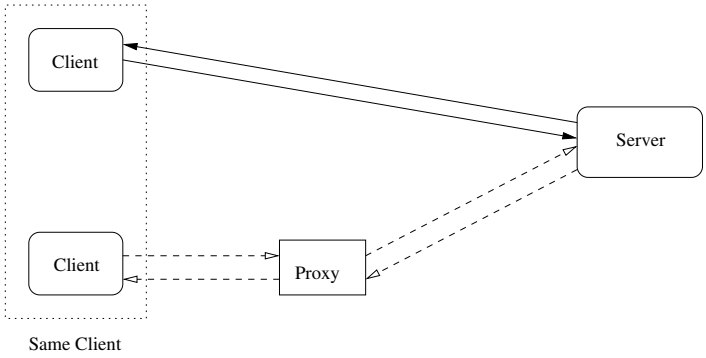

Figure 1. Basic measurement structure for processing overhead of dynamic contents in the proxy.

is a potential disadvantage for using workloads of dynamic contents in our measurements. Dynamic contents often need timely services from the Web server, which may take up to multiple seconds for running service programs. Such dynamic changes and long delays may significantly disturb the server's load, and thus the measurement accuracy. For example, when we access the cgi-bin Web pages, the server needs to fork a process/thread to execute the program, and then sends the result back.

Considering the nature of the processing overhead in the proxy, we have found that the overhead is independent of dynamic contents. The Squid proxy processes a dynamic document exactly the same as it does a static document, if it is the first time to be requested, except that the proxy marks the dynamic document as 'private' for a future replacement (see Section 2.1). Thus, the processing overhead can be accurately measured by using static workloads.

In our experiments, the response time of static Web contents was much more stable and shorter (in the order of 0.1 s), and the Web server was not involved after the content was delivered. In addition, instrumentation in the proxy for measurements can generate additional overhead, possibly disturbing the measurement accuracy.

In our new experiments, the 'client' program periodically sends requests to front pages of a set of selected and representative Web sites that are listed in Table I. The selection is based on the following considerations. First, the selected Web sites are frequently accessed, it is likely that their front pages are always cached in the servers' memories, and the servers are sufficiently powerful to react to a huge amount of accesses. Thus, periodically sending requests to each Web site, we are able to obtain a relatively stable response time. Second, four popular Web site types are covered in our experiments: '.com', '.edu', '.gov', and '.org'. Finally, considering the distance, we selected Web servers on the east coast (www.ets.org, www.ieee.org, www.mit.edu, and www.whitehouse.gov), on the west coast (www.hp.com, www.intel.com, www.microsoft.com, and www.stanford.edu), and a local site (www.wm.edu).

The experiments were set up by running the Squid proxy (version squid2.3-STATBLE4) and client programs on a Pentium 3 Intel 1 GHz processor machine with Redhat 7.1 Linux. The machine

Table I. The selected Web sites and measured average overheads for processing dynamic contents in the proxy.

Site names	Length (bytes)	Overhead (s)	Variance	Standard deviation	Locations
MIT.EDU	6919	0.094	0.025	0.158	MA
STANFORD.EDU	10 197	0.118	0.010	0.102	CA
ETS.ORG	18 903	0.131	0.009	0.093	NJ
WM.EDU	19 160	0.117	0.001	0.033	VA
MICROSOFT.COM	23 167	0.265	0.0003	0.005	WA
IEEE.ORG	26 839	0.260	0.060	0.240	NJ
WHITEHOUSE.GOV	27 655	0.271	0.11	0.33	DC
INTEL.COM	36 831	0.273	0.003	0.055	CA
HP.COM	46 180	0.299	0.078	0.279	CA

was dedicated to the experiments. We minimized possible system intrusion when we measured the processing overhead for two reasons. First, a proxy is normally shared by multiple clients with context switching overheads. In contrast, our proxy serves only one client, minimizing the effect of unrelated overhead in the measurement. Second, in our experiments, the client and the proxy are co-located on the same machine, eliminating the networking transfer time between a client and the proxy. In practice, this networking time can potentially disturb the measurement of the processing overhead.

4.2. Quantifying the processing overhead for Sheltering

In order to cover the entire time period of a day, we conducted the measurement every hour, 24 times a day. Besides the differences in type and distance, the front pages of the selected Web sites had different content lengths. We repeated measurements 100 times for each site to calculate the average Squid proxy processing overhead. In our calculation, we discounted extremely large values that are not possible, and discounted the measured values when some of the Web sites were temporarily unavailable. Table I presents the content length, average processing overhead time, its variance, and the standard deviation of the measurements.

The measured processing overhead of each site is quite consistent, ranging from 0.1 to 0.3 s. The average overhead time is 0.2 s. The quantum of the time overhead accounts for 10–30% of the response time for directly accessing the Web server for a dynamic document, and 3–10 times higher than that for accessing a static document [6]. In order to verify that the measured results are machine independent, we ran the same experiments on an Intel Pentium 4 with 1.7 GHz processor, with the other configurations exactly the same as in the Pentium 3 machine. We obtained almost identical results.

In addition, space overhead is also involved because memory and disk are used to temporarily store the dynamic contents. Besides the required space for the contents, related data structures are allocated, which can be complicated. For example, the structure of *StoreEntry* is used, which includes other complex structures, such as *MemObject*, *HttpReply*, and *HttpHeader, HttpBody*, in turn.

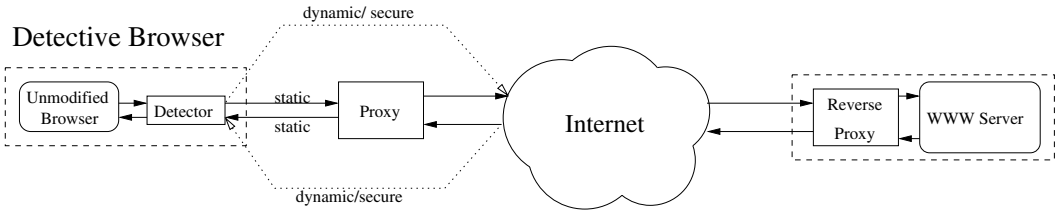

Figure 2. The detective browser model.

4.3. Processing overhead for tunneling

It is difficult to get secured transaction workloads for experiments. Thus, we are unable to provide the measured overhead in this paper. Comparing operation differences, the tunneling overhead should be slightly lower than the sheltering overhead but at a comparable level, where no further request/reply header parsing is necessary. Our major concern of tunneling is not the performance overhead, but the security problem.

5. THE DESIGN AND IMPLEMENTATION OF DETECTIVE BROWSERS

Figure 2 presents the position of the detective browser on the Internet, which consists of an unmodified browser and its attached detector. Upon a client request, the detective browser first checks if the request is for a dynamic document. If so, the request will be directed to the targeted server, bypassing the proxy. Otherwise, the request will be routinely sent to the proxy. In Figure 2, the proxy is set on the client side (client-side proxy or proxy), and/or the server side (server-side proxy or reverse proxy). We try to eliminate the client-side proxy overhead.

5.1. The types of dynamic contents and secured transactions to be detected

Generally, dynamic Web contents have the following features: (1) documents are changed upon each access (e.g. cgi binaries [17], asp [18], fast-cgi, ColdFusion etc.); (2) documents are the results of queries (e.g. the google search engine); and (3) documents embody client-specific information (e.g. cookies [19]) or Server Side Includes (SSIs). Generally speaking, these documents are the following types of dynamic contents: queries, SSIs, and scripts.

- *Scripts*. Scripts are written and executed in different ways. Generally, they could be in the following formats.
  - — cgi (Common Gateway Interface [17]): this is a standard for interfacing external applications with information servers, such as HTTP or Web servers. A plain HTML document that the Web daemon retrieves is static, which means it exists in a constant state: a text file that does not change. A CGI program, on the other hand, is executed in

real time, so that it can output dynamic information. Generally, it can be used to connect a Web server with a wide range of applications. It can be written in different languages, as long as they are executable. For example, the script written by Perl is always labeled with 'pl' as its extension.

— asp (Active Server Page [18]): the operations on asp page are done at the Web server. After the ASP codes are executed, all the asp code is stripped out of the page. A pure HTML page is all that is left and will be sent to the browser.

— PHP (PHP: Hypertext Preprocessing [20]): this is a general-purpose scripting language that is especially suited for Web development and can be embedded into HTML. Like the asp, the code is executed on the server and the client receives the results of running the script.

- *Queries*. The contents in all the search engines belong to this category. Users normally interact with the server by inputting some information into the form (for example, use 'google' to search something by inputting the key word). Normally the server is connected to some background databases, so that the query can be executed and the result can be sent back to the user via the server. Queries can be implemented by forms, CGI, ASP, PHP, JSP, etc. No matter how the queries are implemented, a '?' appears in the URL when a client sends the request.
- *SSIs*. These are applied to an HTML document. They provide for interactive real-time features such as echoing current time, conditional execution based on logical comparisons, and others. An SSI consists of a special sequence of tokens on an HTML page. As the page is sent from the HTTP server to the requesting client, the page is scanned by the server for these special tokens. When a token is found the server interprets the data in the token and performs an action based on the token data. The pages with 'shtml' as a name extension are the SSIs, but some do not have a 'shtml' name extension.

The detective browser is also able to detect the following requests for secured transactions.

- Secure port HTTP requests: when port 443 or 563 is given in the request following the host, then it is clearly a request for secured service from the server. 443 is for secured http, 563 is for snews[¶].
- HTTPS requests: all Netscape versions support the https requests, which is a secured http request, and is done on the SSL. An example is that when a client pays bill on-line, the Web site automatically leads the client to the https instead of http.

The detective browser detects each type of dynamic content and secured transaction as follows.

- For scripts, there are the following.

  — For cgi, the URL must include the 'cgi-bin', and the script ends with the name extension '.cgi' or '.pl'. It will include the symbol '?' when it is used for queries. The detective browser can easily determine the type by parsing the unique symbols.

[¶]The TSL is working to make the secured and unsecured services to share a common port, such as 80.

  - — For asp, all asp pages must have the extensions '.asp', which is easy to check for in the URL. Also, when it is used for queries, '?' must appear in the URL.
  - — For PHP, all PHP pages must have the extension '.php', which is also very easy to check for in the URL. As for asp, when it is used for queries, the '?' must appear in the URL.

- For queries, one or more keywords are always associated with each query. No matter how they are implemented, there must be a '?' in the URL followed by some keywords, so that we can simply check for this symbol in the URL. This can also be combined with searching for 'cgi-bin'.
- For the SSI, we only process the pages with '.shtml'. Since they all have the name extension '.shtml', they are easily detected in the URL.
- For secured transactions, there are the following.
  - — For the HTTPS request, the https will be easily checked out on the URL since 'https' will appear.
  - — For requests to ports 443 or 563, the port number must appear after the URL's host, so it is easily to check it out in the URL.

We have implemented the detector associated with a text-based browser for conveniently measuring its overhead. The detector intercepts the HTTP requests before they are sent out, and then analyzes the requests. A major component of the detector is the StringSearch function for searching the specific symbols representing dynamic contents in the URL or header. If such symbols are detected, the request will bypass the proxy. Another component is the ConnectionRedirect function for bypassing the proxy. We have also implemented it on the Mozilla.0.9.7. (Currently it works on Linux Redhat 7.1.) It is very easy to patch the current standard browser (Netscape/Mozilla) so that it is capable of performing the detection function.

The detector adds some processing time to each request. Our experiments show that the overhead is negligible, as it is only about 1% of the client response time. More details can be found in [21].

6. DETECTIVE BROWSER PERFORMANCE ANALYSIS

If there are not many dynamic requests, or secured transactions, why should we be bothered to make the patch on the browser? To quantitatively determine how effective the detective browser is, we analyzed access traces from NLANR [16]. The time period ranged from 25 February to 4 March, 2002. Among the nine different proxy sites from NLANR, we chose three covering the east coast, the Rocky Mountain area and the west coast of the U.S.A. Traces of the east is from proxy site 'pb.us.ircache.net' located in Pittsburgh, Pennsylvania (simplified as PB). Traces of the Rocky Mountain area is from proxy site 'bo.us.ircache.net' located in Boulder, Colorado (simplified as BO). Traces of the west is from proxy site 'sj.us.ircache.net' located in San Jose, California (simplified as SJ). In the following context, for brevity, we show the analysis on PB Squid proxy as an example, while other results can be found in [21,22].

6.1. The analyzed results from the traces

Table II gives breakdowns of different types of requests to the PB Squid proxy. We put SSIs and scripts together here, since we will give their detailed breakdowns later. The table shows that the sum of the

Table II. The breakdowns of requests to PB.

Date	Total	No. queries (%)	No. SSIs+scripts (%)	No. security (%)
25 February	1 286 520	221 232 (17.20)	48 628 (3.78)	9114 (0.71)
26 February	1 421 559	245 162 (17.25)	51 620 (3.63)	10 271 (0.72)
27 February	1 299 109	241 427 (18.58)	53 631 (4.13)	9732 (0.75)
28 February	1 182 899	175 237 (14.81)	38 456 (3.25)	6738 (0.57)
1 March	998 905	101 228 (10.13)	25 220 (2.52)	6306 (0.63)
2 March	592 992	51 231 (8.64)	15 001 (2.53)	3418 (0.58)
3 March	615 945	50 544 (8.21)	16 196 (2.63)	3751 (0.61)
4 March	1 026 297	113 478 (11.06)	32 607 (3.18)	9263 (0.90)

Table III. The breakdowns of queries to PB.

Date	Total	No. CGI-Q (%)	No. ASP-Q (%)	No. PHP-Q (%)	No. PL-Q (%)	Others (%)
25 February	221 232	17 876 (8.08)	17 124 (7.74)	9970 (4.51)	3208 (1.45)	78.22
26 February	245 162	16 968 (6.92)	23 726 (9.68)	9852 (4.02)	3698 (1.51)	77.87
27 February	241 427	18 594 (7.70)	18 483 (7.66)	10 682 (4.42)	3505 (1.45)	78.77
28 February	175 237	12 127 (6.92)	13 685 (7.81)	6703 (3.83)	2149 (1.23)	80.22
1 March	101 228	10 073 (9.95)	9359 (9.25)	7078 (6.99)	1052 (1.04)	72.77
2 March	51 231	4810 (9.39)	5362 (10.47)	2116 (4.13)	696 (1.36)	74.66
3 March	50 544	5334 (10.55)	4796 (9.49)	1674 (3.31)	548 (1.08)	75.56
4 March	113 478	10 791 (9.51)	8737 (7.70)	5774 (5.09)	934 (0.82)	76.88

queries, SSIs and scripts occupies a high percentage of the total requests, ranging from 11–23%, which can be bypassed from the proxy. In other words, the load on this proxy could be reduced by 11–23% when the detective browser is applied. Table II also shows that the number of requests for secured transactions is small. The main reason for this is that since 1998, the IRCACHE has stopped accepting the SSL requests. Those recorded by the access.log of squid are only those requests to 443 port. This has been further verified by our trace analysis on denied requests in the corresponding access logs and store logs. The total number of detectable requests should be much higher than the number we have reported here, especially as today there are more and more e-commerce services on the Internet.

A very important portion of all the trace logs is to further analyze the queries, SSIs and scripts to ascertain different ways to implement them. The breakdowns are especially meaningful in reflecting which methods are widely used and which are not widely accepted by the clients.

Table III shows the breakdowns of the queries to the proxy PB. The data indicate that ASP and CGI are used more frequently than PHP and PL in the query requests to the PB proxy. But the major parts are from other methods, such as forms.

Table IV. The breakdowns of the SSI and Scripts to PB.

Date	Total	No. SHTML (%)	No. CGI (%)	No. ASP (%)	No. PHP (%)	No. PL (%)
25 February	48 628	3403 (7.00)	16 450 (33.83)	12 688 (26.09)	9655 (19.85)	6432 (13.23)
26 February	51 620	4312 (8.35)	12 865 (24.92)	14 096 (27.31)	8476 (16.42)	11 871 (23.00)
27 February	53 631	4268 (7.96)	14 851 (27.69)	14 712 (27.43)	10 588 (19.74)	9212 (17.18)
28 February	38 456	2995 (7.79)	7873 (20.47)	11 186 (29.09)	7200 (18.72)	9202 (23.93)
1 March	25 220	2106 (8.35)	5683 (22.53)	8583 (34.03)	5481 (21.73)	3367 (13.35)
2 March	15 001	1325 (8.83)	3740 (24.93)	4753 (31.68)	2829 (18.86)	2354 (15.69)
3 March	16 196	1333 (8.23)	3808 (23.51)	5754 (35.53)	3002 (18.54)	2299 (14.19)
4 March	32 607	2184 (6.70)	6976 (21.39)	10 697 (32.81)	9084 (27.86)	3666 (11.24)

Table IV illustrates that CGI and ASP are the commonly used scripts and the SSIs are not widely used at present—only about 10% are composed of SSIs.

6.2. What is the detective browser unable to detect?

Besides the four types of common dynamic contents (cgi, queries, asp, and cookies), the detective browser can also detect the following two dynamic content types: (1) Method (the request method other than 'GET' and 'HEAD'), and (2) Auth (a request with an authorization header). However, the detective browser is not able to process the following uncacheable Web contents, since they are only designated by the Web servers' response.

- Pragma: the response is explicitly marked uncacheable with a 'Pragma:no-cache' header.
- Cache-control: the response is explicitly marked uncacheable with the HTTP 1.1 cache-control header.
- Response-status: the server response code does not allow the proxy to cache the response.
- Push-content: the content type 'multipart/x-mixed-replace' is used by some servers to specify dynamic content.
- Vary: the vary is specified in the header.

The usage of the above dynamic content types is low (the percentage). We believe there may be some other rare requests that are not well filtered by the current version of the detective browser. The detective functions will be upgraded as the formats of dynamic contents and secured transactions are updated.

7. CONCLUSION

We have identified and quantified two overhead sources in the proxy for processing dynamic Web contents and secured transactions. We have also shown that this overhead source could not be easily eliminated from the proxy, and security concerns can be serious for the proxy to tunnel secured transactions. Avoiding the delay caused by proxy processing overhead for accessing dynamic contents,

and addressing the security concerns, our detective browser actively determines if a request should go directly to the Web server bypassing the proxy, or go through the proxy. We have shown the effectiveness of this approach, and its low overhead in implementations.

ACKNOWLEDGEMENTS

We would like to thank for the editor and referees for their constructive feedback on this paper.

REFERENCES

1. Challenger J, Iyengar A, Dantzig P. A scalable system for consistently caching dynamic Web data. *Proceedings of INFOCOM*, New York, March 1999. IEEE Computer Society Press, 1999.
2. Smith B, Acharya A, Yang T, Zhu H. Exploiting result equivalence in caching dynamic Web content. *Proceedings of the Second USENIX Symposium on Internet Technologies and Systems*, Boulder, CO, October 1999. USENIX Association Press, 1999.
3. Yin J, Alvisi L, Dahlin M, Iyengar A. Engineering server-driven consistency for large scale dynamic Web services. *Proceedings of WWW*, Hong Kong, May 2001. ACM Press, 2001.
4. Holmedahl V, Smith B, Yang T. Cooperative caching of dynamic content on a distributed Web server. *Proceedings of the 7th IEEE International Symposium on High Performance Distributed Computing*, Chicago, IL, July 1998. IEEE Computer Society Press, 1998.
5. Iyengar A, Challenger J. Improving Web server performance by caching dynamic data. *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997. USENIX Association Press, 1997.
6. Zhu H, Yang T. Class-based cache management for dynamic Web content. *Proceedings of INFOCOM*, Anchorage, AK, April 2001. IEEE Computer Society Press, 2001.
7. Candan KS, Li W-S, Luo Q, Hsiung W-P, Agrawal D. Enabling dynamic content caching for database-driven Web sites. *Proceedings of ACM SIGMOD*, Santa Barbara, CA, May 2001. ACM Press, 2001.
8. Cao P, Zhang J, Beach K. Active cache: Caching dynamic contents on the Web. *Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, The Lake District, U.K., September 1998. Springer, 1998.
9. Luo Q, Krishnamurthy R, Li Y, Cao P, Naughton JF. Active query caching for database Web servers. *Proceedings of the 3rd International Workshop on the Web and Databases*, Madison, WI, June 2000.
10. Douglis F, Haro A, Rabinovich M. HPP: HTML macropreprocessing to support dynamic document caching. *Proceedings of USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997. USENIX Association Press, 1997.
11. SSL 3.0 Specification. http://www.netscape.com/eng/ssl3/ [April 2001].
12. Squid Web Proxy Cache. http://www.squid-cache.org/ [April 2001].
13. Yoshikawa C, Chun B, Eastham P, Vahdat A, Anderson T, Culler D. Using smart clients to build scalable services. *Proceedings of USENIX Annual Technical Conference*, Anaheim, CA, January 1997. USENIX Association Press, 1997.
14. Ralf S. Engelschall. http://www.openssl.org/ [April 2001].
15. http://realprogrammers.com/hack/SSH/upload_code_tunnel_out.html [April 2001].
16. IRCache. http://www.ircache.net/ [April 2001].
17. Gate Interface. http://hoohoo.ncsa.uiuc.edu/cgi/overview.html [April 2001].
18. Active Server. http://www.takempis.com/asp1.asp [April 2001].
19. Netscape. http://home.netscape.com/newsref/std/cookie_spec.html [April 2001].
20. PHP Group. http://www.php.net/ [April 2001].
21. Chen S, Zhang X. Detective browsers: A software technique to improve Web access performance and security. *Proceedings of the 7th International Workshop on Web Content Caching and Distribution*, Boulder, CO, August 2002.
22. HPCS Lab, College of William and Mary. http://www.cs.wm.edu/hpcs/ [April 2001].

