Software Issues in Characterizing Web Server Logs

Balachander Krishnamurthy and Jennifer Rexford

AT&T Labs–Research; 180 Park Avenue
Florham Park, NJ 07932 USA
{*bala,jrex*}*@research.att.com*

1 Introduction

Web server logs play an important role in measuring the overhead on servers and the network, as well as in evaluating the performance of features of the HTTP protocol. Presently, there are several products on the market that analyze logs, employing a variety of techniques to store and manipulate them. For example, Accrue [1] uses a relational database, Andromedia [2] uses an object oriented database, while Netgenesis [3] uses Informix. Other commercial log analyzers include Sawmill [4], SurfReport [5], and WebTrends [6]. These companies do not go into detail about the mechanisms they use to clean and process the logs for obvious reasons. Most researchers and academicians have access to logs from a few sites. These logs range in durations from a day, to a few weeks, to several months. The number of hits on sites vary from few hundred thousand to several million.

Processing large and varied server logs introduces a number of important software challenges, including:

- **Processing overheads**: Servers logs typically have information about large numbers of clients, requests, and resources. This can introduce substantial computational and memory overheads in processing the data.
- **Data integrity**: Entries in server logs sometimes include erroneous or inconsistent information that should be omitted or cleaned. In addition, requests for a single resource may have multiple URL representations that need to be unified.
- **Privacy and security**: Server logs typically reveal potentially sensitive information about the requesting clients and requested resources, such as temporal distribution and frequency of requests from clients, duration of time spent at the site, and the nature of the resources being requested[1].

To address these issues, we propose a process for cleaning and anonymizing server logs, and producing a simplified intermediate format for post-processing. Though we restrict the discussion to server logs, most of the comments apply more broadly to proxy and client logs as well.

We describe the process in the context of our research on efficient ways for Web servers to provide hints to proxies and clients about future accesses [9, 8]. Some of the logs used in these studies are presented in Table 1. AIUSA is from Amnesty International USA's web site log, Marimba is from Marimba Corporation, Apache is from the popular web server site, and Sun is from Sun Microsystems. The EW3 logs are a collection of four of the larger server logs from AT&T's Easy World Wide Web hosting service; EW3 currently hosts approximately 8,200 sites [12]. Nagano is IBM's 1998 Winter Olympics log, which was aquired recently and has not yet been "cleaned."

[1]Sports scores, Amnesty International torture reports, user's search strings, or pornographic material in Government sanctioned reports, etc. Additionally, if cookies and session identifier information are present, individualized information can be easily tracked.

Server Log (days)	Number of Requests	Number of Clients	Unique Resources
AIUSA (28)	180,324	7,627	1102
Marimba (21)	222,393	24,103	94
Apache (49)	2,916,549	271,687	788
Sun (9)	13,037,895	218,518	29436
EW3_4 (94)	6,273,792	79,623	2638
EW3_6 (94)	6,281,303	102,454	784
EW3_18 (94)	3,902,302	58,839	279
EW3_60 (94)	45,903,646	481,628	2,031
Nagano (1)	11,665,713	61,707	33,875

Table 1: Some server logs and their characteristics

Given the range and diversity in the collection of logs, we need robust and efficient tools to clean and process the logs—we relied on the *libast* library and the *sfio* (safe/fast I/O) routines [10, 11]. The primary goal of the *libast* was to increase reuse and portability, while *sfio* provided ways for efficient manipulation of buffers. These two libraries and other more efficient and correct implementations of several popular UNIX commands are part of the *ast* collection [10].

2 Cleaning the Server Logs

As part of processing an HTTP request, the Web server generates a log entry with several fields in it; the number of fields range anywhere from half a dozen to twenty elements depending on the server). There are over a dozen different logging formats including variations on common logging formats, such as Apache's ECLF (Extended Common Log Format), which has additional fields. Some of the key fields found in most logs include:

- IP address or name of the client (remote host)
- Date and time of the request
- First line of the request including the HTTP method and URL
- HTTP response status code (200, 304, ...)
- Number of bytes in the response

In addition, logs might have the remote log and user's name (rarely present), the referer field—the URL from which the current was reached (found occasionally), user agent information—OS and browser version used (found sparingly). Although these fields are typically assigned and printed correctly, individual entries may become corrupted, depending on the robustness of the logging mechanism and the I/O subsystem. For example, the log may include incorrect values for fields that were not populated by the server. Or, entries may have extra or missing fields if multiple server threads (each processing a different HTTP request) compete to print log entries without sufficient locking mechanisms. As a result, large server logs often have errors.

Many of these errors can be detected through conventional defensive programming techniques. For example, our routine for reading the server logs checked whether each entry had the expected number of fields (e.g., by checking the return value of `scanf`). Entries that violated this check were manually inspected. Although most of these entries were deleted, a few cases involved URL strings that contained embedded newline characters, which caused the `scanf` to mistakenly detect the end of a line; these entries were edited to remove the offending control characters. For the entries with the appropriate number of fields, we verified that each field had the correct type and range of values. For example, timestamps had to fall within the collection period for the server log, and HTTP response codes had to lie within the small set of acceptable values. Entries with invalid fields were manually inspected and removed. The Sun log in Table 1 was shortened by about 5% after being cleaned.

Processing URLs introduced a number of challenges. First, URLs can be arbitrarily long. To avoid having to guess a maximum length for a URL, we read the server logs using the *sfio* (safe/fast I/O) library's `sfgetr` function, which automatically allocates the appropriate amount of memory for each string. Second, URLs have a wide range of acceptable formats, leading to multiple representations for the same resource. For example, `http://www.xyz.com/foo.html` may also appear in the server logs as `www.xyz.com/foo.html`, `foo.html`, or `www.xyz.com///foo.html` (as well as variations with embedded newline characters, as discussed above). We canonicalized the URL by deleting leading `http://` and the site name, any extra `/` characters, via a fast regular expression routine in *libast*. While this process resolves many of the URLs, some cases are still difficult to handle. For example, `http://www.xyz.com/bar` could refer to a resource `bar` or could actually resolve to `http://www.xyz.com/bar/index.html` if `bar` is a directory.

3 Constructing Concise Representations

Rather than using the cleansed logs in our performance studies, we converted each log into a sequence of integer tuples. Each client and each URL were associated with a unique integer identifier, and timestamps were converted to an integer (Julian seconds since start of epoch). This representation avoids revealing unnecessary information about the requesting clients and the requested resources. In addition, the tuple format provides a single representation across a range of server logs, which may record different set of fields and have different ways of representing time. The `tmscan` routine in *libast* is capable of handling virtually any date format. The tuple format also reduces the size of the logs, by reducing the number of fields and the size of each field, and avoids the need to deal with variable-length strings in the rest of the code.

The tuple representation was constructed with a single pass through the clean logs, using two hash tables to store the unique identifier assigned to each client and URL string. After experimenting with a conversion program written in awk/Perl, we realized that the processing and memory requirements were very high, particularly for server logs with millions of requests for tens of thousands of different resources. Using the hash routines in *libast* and a small C program significantly sped up the effort[2]. Some aspects of the tuple construction were specific to our study of ways for servers to provide hints about future accesses. For example, we were interested in grouping resources that have the same directory prefix in their URLs (e.g., `foo/foo.html` and `foo/bar.html`) to determine if these resources were typically accessed together. So, our tuple representation also included integer identifiers for each one-level and two-level directory prefix. Similarly, we can group clients based on the IP net or subnet, to project how our schemes perform when related clients access the server through a single proxy. This requires hashing on portions of the client address field in the server log, and performing a DNS look-up for log entries that provide the client machine name instead of the IP address.

The tuples could also contain additional information derived implicitly from the log fields, such as the content type of the resource (e.g., classifying URLs with a `.jpg` or `.gif` suffix as images) and whether the resource is dynamically generated (e.g., a URL with a `cgi` or `?`). Again, the regular expression routines in *libast* came in handy. Finally, depending on the application, it may be possible to limit the number of unique identifiers, to avoid memory and computational complexity in later stages. Many resources are accessed very infrequently; resource popularity follows a Zipf's law [7]. Thus it might be useful to focus attention on frequently requested resources alone. Likewise, many of the requests may come from a few clients and it might be useful to restrict attention to this subset of clients. In our study of server prediction schemes we evaluated techniques where server response messages include hints about future accesses. Given the computational

[2]The awk script ran for 14 hours on the Sun log before termination by human intervention. The C program completed in a couple of hours. Memory consumption appeared to be the main culprit.

complexity of constructing accurate hints for a resource, it made sense not to piggyback hints on responses for unpopular resources. To reduce memory overheads (array sizes) in evaluating the prediction schemes, we assigned a single resource identifier to all resources below a certain popularity threshold.

To identify the unpopular resources, we generated a list of the unique resources and their access frequencies, and chose a threshold for identifying unpopular resources. The threshold will vary with application and thus should be a parameter for each analysis. We followed a similar approach to rank the various clients that contacted the server. The really high volume clients required closer examination: for one of the logs, the top client was a spider; in another log, the top client was an internal site used to update and change the content at the Web server. For our study, we removed requests from these clients since the access patterns would not be representative of the intended users of the site. We realize however that such inferences cannot be automatically gleaned but it is important to be aware of them, since blind studies could result in skewed statistics.

4 Collecting Performance Metrics

After converting the clean logs into a tuple format (with fields for the client, time, resource identifier, and two levels of URL directory prefixes), we processed the tuple log to collect various performance metrics. To measure client access patterns, we needed to process the set of log entries for each individual client. Rather than processing the tuple log in a time order, and keeping separate statistics for each client, we first sorted the log by the client identifier. This allowed us to focus on one client at a time, and then accumulate the overall statistics after processing each client. To keep client requests in the appropriate order, we sorted entries for the same client based on the time field. Correctness required a *stable* sort to ensure that entries with the same client and same timestamp stayed in the right order (e.g., requests for a page and its embedded images often occurred within the same one-second period). Depending on the installation, the UNIX `sort` command often does not perform a stable sort by default; in some implementations it can be specified as an option. We used the *sort* in the *ast* collection which provides stable sorting as default.

5 Conclusion

Our experiences cleaning, converting, and analyzing a collection of server logs have taught us a number of valuable lessons about dealing with large Web datasets. In cleaning the logs, we saw that server logs often have errors and inconsistencies, requiring defensive programming, and some manual intervention. Dealing with Web server logs is complicated by the fact that URLs can be quite long and have a range of acceptable formats. In converting the logs to a tuple format, we realized that using hash functions to convert strings to integer representations offers a substantial reduction in processing and memory complexity in the rest of our study, and avoided the need for other researchers to work with the original log files. This was very helpful in separating the software development for cleaning and converting the server logs from the code that computed server hints and evaluated the effectiveness of our prediction scheme. Also, identifying unpopular resources and unusual clients proved useful in focusing our study on typical client access patterns.

Analyzing the data was simplified by sorting and post-processing the tuple logs, rather than writing a simulator that would sequence through the log entries in time order. This process was simplified by the use of stable sorting, and by storing intermediate results. This enabled us to generate predictions and collect performance metrics by performing just two passes through the sorted tuple log. This separation of the software was helpful in scaling our study to a large number of logs and parameters. Cleaning the logs and converting to a tuple representation could be performed once for each log, whereas the construction of server hints was performed for several sets of parameters, and performance metrics were collected over a wide range of configurations. Finally, throughout all of the stages, we found it extremely useful to draw on existing library support for file I/O, hash tables, and regular expressions. This enabled us to write robust and efficient C programs, without sacrificing the simplicity and flexibility available in languages like awk and Perl.

Acknowledgements: We thank Glenn Fowler and Phong Vo for their software and comments, and we thank Anja Feldmann and Albert Greenberg for their comments on an earlier version of the paper.

References

[1] http://www.accrue.com/.

[2] http://www.andromedia.com/.

[3] http://www.netgenesis.com/.

[4] http://www.flowerfire.com/sawmill/.

[5] http://www.netrics.com/SurfReport/.

[6] http://www.webtrends.com/.

[7] Martin F. Arlitt and Carey L. Williamson. Internet Web servers: Workload characterization and implications. *IEEE/ACM Trans. on Networking*, 5(5):631–644, October 1997. `ftp://ftp.cs.usask.ca/pub/discus/paper.96-3.ps.Z`.

[8] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Efficient algorithms for predicting requests to web servers, July 1998. In submission. `http://www.research.att.com/~bala/papers/inf99-submit.ps.gz`.

[9] Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford. Improving end-to-end performance of the web using server volumes and proxy filters. In *Proc. ACM SIGCOMM*, September 1998. `http://www.research.att.com/~bala/papers/sigcomm98.ps.gz`.

[10] Glenn Fowler, David Korn, and Kiem-Phong Vo. Libraries and file system architecture. In Balachander Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 2. John Wiley & Sons, New York, NY, 1995. `http://www.research.att.com/library/books/reuse`.

[11] David Korn and Kiem-Phong Vo. Sfio: Safe/fast string/file IO. In *Proceedings of Summer 1991 USENIX Conference*, pages 235–256, 1991.

[12] AT&T Easy World Wide Web. `http://www.att.com/work-net/easywww`