servload: Generating Representative Workloads for Web Server Benchmarking

Jörg Zinke, Jan Habenschuß and Bettina Schnor
Institute for Computer Science, University of Potsdam, Germany
Email: {jozinke, schnor}@uni-potsdam.de

***Abstract*—Web server benchmarking has two main purposes, comparing different hardware platforms or server configurations, and testing the scalability of a given system.**

For the first purpose, the capability to replay given workload traces is needed. For the second purpose, also the capability to generate increased *representative* workloads is needed. Scalability tests are of special interests for Internet Service Providers (ISPs) to estimate performance numbers for Service Level Agreements (SLAs) contracted to customers. Depending on the setup and the implemented web application the required scalability may vary from a few requests per seconds up to a waste amount of requests.

This paper presents a web server benchmark called *servload* which generates modified workloads from given traces keeping similar characteristics. *servload* is generic in the sense that it is not dedicated to a special application field. It can process traces from different application fields for example banking, e-commerce or collaborative wikis.

The different workload generation methods of servload are discussed and evaluated. In the presented experiments *servload* shows good replay capabilities.

***Index Terms*—Web Server Performance, Benchmark, Workload Characteristics, HTTP Replay, servload**

I. Introduction

Benchmarking of web servers is required for Internet Service Providers (ISPs) to estimate performance numbers for example determine the number of concurrent users their systems are dimensioned for. This enables them to satisfy guarantees and Service Level Agreements (SLAs) contracted to customers. Such contracts usually contain ascertained response times for web services and web sites too. Especially selective promotion campaigns can lead to high web traffic from a lot of users resulting in slow response times and missed SLAs.

For benchmarking web servers recorded web server logs from different workload situations and campaigns can be taken into analysis and used to simulate similar workloads within a replay of the requests. This can help to understand and reproduce the reachability of the web server from the users point of view. Replaying of recorded web server logs could provide same load on the web servers but is limited as web server logs usually do not provide all required information to reproduce an exact replay. Replaying workloads on different web servers gives the option to compare their capabilities. Compared replay of web server logs requires preprocessing and analysis followed by performance benchmarks.

The second step within preprocessing after the analysis could be the modification of the web server logs to generate higher loads and determine capability limits of the web servers and prove guarantees and SLAs. Keeping workloads representative while generating and modifying them requires suitable analysis methods to gather behavior and sessions of users. If the generated workload conforms to realistic user behavior they are suited for determine capability limits of production web servers.

For large web site a single web server is not enough to handle all incoming requests. ISPs of such web sites usually cluster multiple web servers and put load balancers in front of them to distribute the incoming requests to the web servers. Such Server Load Balancing (SLB) increases the availability and makes it possible to scale the web site with upcoming requirements of more requests.

The need to test the authors load balancing algorithms derived from Self adapting load balancing network (salbnet) [1], [2] was the main motivation for the design and implementation of the proposed web server benchmarking *servload*.

The next section presents important requirements for workload generation and replay. In section III, the related work is discussed. The section IV presents metrics which are suited to characterize *representative* workloads. Based on these metrics, the score method is developed and presented in section V. The score method generates increased or decreased workloads based on existing web server logs. In Section the requirements for a realistic replay are described. Section VII introduces *servload* which implements the lessons learned from the previous sections. Section VIII presents an evaluation of *servload*.

II. Requirements

A. Requirements for Workload Generation

Real Workload: A real workload is defined as a sequence of requests which was received from a real world production web server.

Representative Workloads: A representative workload is defined as a (generated) workload which has the same characteristics like a given real workload. For the characterization of workloads, session metrics like the number or requests, the median think time, or the session length in seconds are proposed (see section IV-B).

For example, a representative workload is the replay of a web server log. In case of a web server cluster, the generation of a representative workload requires the handling of log files from multiple web servers and in different formats like the combined log format and the common log format.

For scalability tests, increasing a workload is also required. The generated workload should be representative again. Therefore, workload generation should be able to identify and analyze sessions and implement methods to increase or decrease the number of requests and sessions.

B. Benchmark Requirements

The web server benchmark should be able to replay web server logs.

Replay: The replay of web server logs is defined as issuing the sequence of requests found in the logs in the same timed order respecting the original think times between requests. The order is achieved through sorting the requests by their timestamps. The ordered replay of the request sequence should result in a similar log on the web server as the original one.

Furthermore the web server benchmark should provide detailed performance statistics about the number of requests, the number of sessions, throughput, connect time and error counts like the number of connection errors, number of timeouts and possible unexpected Hypertext Transfer Protocol (HTTP) status codes. Instead of the often used transfer time of the response which depends on various parameters like the type of the transferred data and the current up-link of the client, the time until the first byte of the response is received, the so-called *First Response Time* is favored. This metric is also suited for SLAs definitions.

The benchmark should provide a good performance to handle thousands of requests and should also work in environments with web server clusters. Hence, a server load balancer like for example the Linux Virtual Server [3] must be supported.

In addition the benchmark should run on IPv4 and IPv6 and should support HTTP in version 1.0 and 1.1 as well as Hypertext Transfer Protocol Secure (HTTPS) and optionally further protocols like Domain Name System (DNS).

III. Related Work

A. Comparison of Existing Benchmarks

SPECweb2009 [4] is a web server benchmark from the Standard Performance Evaluation Corporation (SPEC). *SPECweb2009* neither supports load balancing clusters nor the replay functionality.

The benchmarks *http_load* [5], *Siege* [6], and *Pylot* [7] ignore think times and different user sessions.

The benchmark *httperf* [8] has support for a loading requests from a sessions file.

Tsung [9] provides complex session configuration features and options and replay of recorded sessions as well as graphical output of the result. The drawback is that the result metrics are aggregated to an unsuited mean value from a 10 second sampled spot. Also the minimum and maximum values of the metrics are taken from the sampled spots and thus may not contain the real minimum and maximum. Interesting metrics like the number of requests which exceeded a threshold response time are not available.

None of these benchmarks is suited to scalability tests, i.e. none is capable to generate increased representative workloads.

B. Generation of Representative Workload

There exist several research projects which investigate the generation of *representative* workload. These concepts are discussed in the following. Practical tools which could be reused are not available from these projects anymore.

In [10] the authors present *SURGE*, a tool which is also dedicated to generate representative web workload. The approach is quite different from ours. The authors discuss web workload characteristics like file sizes, mean think time, and the number of requests made to an individual file (popularity). They use published client data sets from 1995 to derive distributions for file sizes, mean think times, and the number of embedded references. These distributions are used by *SURGE* to generate the workload. Hence, the quality of the generated workload depends on the quality of the modelling done by an expert. While heavy tailed distributions may also be a correct assumption for the file size nowadays, it is questionable how good the parameters still fit. In our approach, however, the workload generator can be fed by any current trace and will generate automatically higher workloads with similar statistically characteristics.

In [11] the author presents the Website Usage Signature (WUS) concept to decide whether a workload is realistic or not. They argue that a load test that does not reflect actual usage is at best useless, and it could be dangerously misleading. The WUS consists of several metrics to measure how closely a load test matches real world usage. Server side WUS metrics are: sessions per hour, page views per hour, hits per hour, average session duration, average page views per session and Average hits per session. While the author demonstrates the difference between the WUS of mean load and peak load, there is no approach presented to create increased workloads for scalability testing.

In [12], the authors present an approach to model realistic user behavior from web server logs using continuous time Markov chains. The authors introduce a hit relation matrix $P(n \times n)$ for all n pages which contains the probability of leaving and returning from and to the site and the probability of switching to a linked page. The goal is to model a realistic so-called *virtual user*. The authors argue that by adding more and more *virtual users*, the scalability of the system can be tested, i.e. one can figure out how many virtual users can be active without reaching the threshold for the response time.

A disadvantage of this approach is that the values in the matrix P are constant which results in the same probability for successor page visits no matter how often this page is requested within a user session. For example, on entering a site it is less likely that the user directly leaves the site, while after some page views leaving becomes more likely. An evaluation whether the generated workload has similar metrics like the given web server logs is not done by the authors.

In [13], [14] statistical models are investigated to generate user sessions that are more or less likely to occur in practice.

They analyze web server logs and model the probability of each possible user session. Sant et al. [13] propose that the probability of a user session can be modeled as the product of the probability of each request within the session. The authors state that it is beyond the scope of their study to analyze generated sessions to see if they match a valid use case. Embedded page elements and session characteristics like the think times of the user or the number of requests per session are not considered and therefore this approach will not create representative workloads.

The formchart model presented in [14] models a web site with a manually created formchart which is a bipartite state machine consisting of pages, actions and transitions between them. A web log analysis is used to determine the probabilities for activities of a page. With the gathered data a decision tree can be built which is theoretically unlimited in its depth and handles cycles based on running indexes. Such a decision tree allows to generate sessions of arbitrary length. Together with further parameters like session length or think times it becomes possible to model different user behavior classes like novice or experienced visitors. There is a graphical implementation available [15] but nevertheless formchart models lack automation and are static thus their creation requires manual interaction and knowledge about the target web site.

The Realistic Usage Model (RUM) presented in [16] is a combination of prepared scenarios and a simple model gathered from web server logs. A scenario corresponds to a use case in Unified Modeling Language (UML) and consists of multiple sessions. Web server logs or user behavior of an existing application can be used to augment the RUM.The distribution of Uniform Resource Identifiers (URIs) in the web server logs can be counted and mapped to the session actions. The URI distribution is also used to determine the probabilities of the *generic sessions*. Furthermore, the authors of [16] implemented the RUM model in a Load Testing Automation Framework (LTAF) which makes scenarios reusable for several web sites as long as the actions map to HTTP requests. This mapping lacks automation and requires manual interaction and knowledge of the web site.

The scenarios can be used to model different user behavior caused by different roles for example administrator or normal user. However, it remains open how to determine the probabilities for different scenarios and how to handle canceled user sessions and therefore this model is not suitable for generating representative workloads.

IV. Metrics

It is required to analyze web server logs to identify and quantify different users and their behavior which is important for generating representative workloads and proper replay.

The format of web server logs limits the analysis as only fields which are available can be considered. The well known *common log format* consists of seven fields: the *remote host* e.g. IP or address of the client, the Request for Comments (RFC) 1413 [17] *identity check* of the client, the user identity of the *remote user* requesting the document as determined by HTTP authentication [18], the *timestamp* when the server finished processing the request, the *request*, the *HTTP response status*, and the size of the response in *bytes*.

The *combined log format* is shown in listing IV.1 and extends the common log format by two tailed fields: *referrer* corresponds to the Referrer-Request-Headers in RFC 2616 [19] and *user agent* corresponds to the user agent HTTP request header. The *referrer* field may reveal the predecessor page of a user as required by models mentioned in section III.

A web server like the popular [20] Apache HTTP server [21] can process multiple concurrent requests through processes and threads which may result in unsorted log entries under high load. In conjunction with the limited resolution of timestamps (a second for example) this may result in log entries with the same timestamp loosing the original arrival order.

A. Metrics for Identifying User Sessions

The *remote user* field can be used for user session identification but is only available on rarely enabled HTTP authentication and otherwise empty. Likewise the *identity check* field could be used for user session identification but is also often not available, for example disabled by default in Apache HTTP server.

Determining the user sessions from the *remote host* field may be inaccurate as multiple users may share the same IP connecting through a HTTP Proxy or are located behind routers which are doing Network Address Translation (NAT). The latter one especially may happen with multiple mobile devices connected to the same cell site as it can occur in the use case which will be presented in section VIII-B.

In case of the combined log, the *user agent* field becomes available and can be considered in conjunction with the *remote host* field to detect requests belonging to the same user session. This may still be improper as users behind the same NAT router may use the same user agent too.

In conclusion identifying user sessions from web server logs requires the following fields to be considered in this prioritized order: *remote user*, *session id*, *remote host* and *user agent*, or *remote host*.

B. Metrics for Identifying User Behavior

The number of requests in a session helps comparing user sessions. Several requests may be the result from a single user action. This metric helps to determine if sessions with a lot of requests or sessions with only few requests are the characteristic user behavior for the given web site within the analyzed time frame.

The think time (in seconds) can be defined as the time between two request arriving at the web server, or as the time between two actions of the user. Both views result in different values as an action may consist of several requests. In the following the time between two arriving requests is used, since this time is easy to determine from the log file. It is recommended to use the median think time to reduce effects

```
127.0.0.1 - sven [22/Oct/2011:16:35:46 -0200] "GET /test.png HTTP/1.0" 200 232 "http://example.com/index.
    html" "Mozilla/5.0 (Windows NT 6.0;)"
```

Listing IV.1: Example for a combined log format entry.

resulting from aborted sessions. The median think time is not defined for sessions which consists of only one request.

The session length in seconds between the first and the last request suffers from the same problem as the median think time: discontinued sessions may not be detected and are included in the total length.

The response body sizes can differ widely resulting from different content types like text and videos. Furthermore, in case of the status code `304 Not Modified` the body size bytes may be zero and the web client will use the version from its cache.

Therefore, for the characterization of the user behavior the following metrics: the number of requests in a session, the session length, and the think time between requests are suited. Furthermore a web server log may provide the body sizes of requests within sessions. Statistical analysis of user session metrics should include mean, median, minimum and maximum values as well as the number of requests per each second of the considered time frame.

Besides the bytes these metrics are protocol independent and can be used for other logs like name server logs as well.

V. Generation of Representative Workload

The methods mentioned previously in section III are based on a sequence of page changes and do not generate a final request sequence representing the workload. With these methods the benchmark need to demand all required page element requests. Furthermore none of the presented methods is able to modify the workload for example increasing the workload is not possible.

According to [22] workloads for the determination of capability limits need to consider aborted sessions as well since users of aborted sessions request less web site resources. Finally, workload generation from web server logs should be done automatically without manual interaction or extra web site knowledge.

In the following three methods to generate workloads are presented, the first two add additional requests, while the third one rates sessions and adds *typical* sessions.

A. multiply *Method*

The *multiply* method increases the number of requests by a given factor.

Each request from a $destination$ time frame is copied and inserted multiple times. The insertion $factor$ is defined in the range of $[1, \infty]$ which results at least in doubling the number of requests:

$$requests_{output} = requests_{destination} * (factor + 1)$$

This method roughens the request sequence with rising insertions as the absolute gap between timestamps with few requests and timestamps with a high number of requests increases.

Since this method replicates every request, also requests from search engine robots are replicated. This may not be representative as robots are usually scheduled and do not increase their visits over the same time frame.

Further, this method does not create new sessions, but only adds requests multiple times within the same session.

B. peak *Method*

The number of request per second usually varies over time frames. The *peak* method tries to increase the highest values within a time frame while the lower values are not touched resulting in a more or less unequally request distribution compare to the *multiply* method. To achieve this first all sessions are identified in the $destination$ time frame afterwards the number of requests for each second is counted and sorted in the same time frame. Based on the results and a $percent$ parameter the threshold for the minimum number of requests per second is determined. The obtained threshold is compared against the number of all requests at the starting timestamp of each session. If the threshold is exceeded the number of requests of the current session is multiplied by a $factor$ from the range $[1, \infty]$ and thus at least doubled. The $percent$ parameter should be configurable in an implementation to allow different increases.

The *peak* method roughens the request sequence as the absolute gap between time frames with few requests and time frames with a high number of requests increases. The method scales with each type of request sequence. Even if the request sequence is distributed equally over a $destination$ time frame the method finds peaks and increases them. Together with the total number of requests per session the number of requests in specific time frames increases and results through the time distances of the requests in a wide variation and a steady boost.

The only disadvantage of the method is that one can not predict the number of added requests as this depends on the original workload and the chosen threshold. Furthermore this method also does not create sessions hence similar to the *multiply* method this may result in irrational request sequences.

C. score *Method*

The *score* method identifies sessions and rates them. Higher rated sessions are *typical* sessions which are going to be used to increase the workload.

The rating is based on the four metrics presented in section IV-B: number of requests, session length, median think time, and median bytes. Sessions from the $source$ time frame are rated for each of the four metrics. Each session metric is

compared to the median of the same metric calculated from all sessions. The nearer the current session metric value is to the overall median the higher is the rating of the session. For each session, the sum of all four metric ratings is calculated. The highest rated sessions will be added first to the $destination$ time frame if the workload is going to be increased. To have wide spread requests created sessions starts at a random position in the $destination$ time frame.

A configurable $percent$ parameter controls the increase:

$$requests_{output} \approx requests_{destination} * percent * factor$$

Hence, the maximum possible increase is a doubling of all sessions and according requests if the $source$ and the $destination$ time frame are the same. Either using a larger $source$ than $destination$ time frame (recommended) or an additional $factor$ parameter to multiply a single session is required to scale increase unlimited. Using a high $factor$ and a low percent value is not recommended as this will result in only the same typical sessions without wider and more realistic variance.

The *score* method is shown in algorithm V.1. The $metric()$ method determines the score for a metric of a session based on the relative distance to the overall median depending on the metric values of all other sessions.

```
1:  /* score for each source session */
2:  for all s in source do
3:      score_s ← 0
4:      /* score for each metric */
5:      for all m in metrics do
6:          score_s ← score_s + metric(source, s, m)
7:  sessions ← sort(source, score)
8:  counter ← count_req(destination) * percent
9:  /* add sessions to output */
10: for all s in sessions do
11:     for i = 0 to factor do
12:         output[] ← output[] + get_req(s)
13:         handle_think_times(output[])
14:     counter ← counter − count_req(s)
15:     if counter < 0 then
16:         break
```

Algorithm V.1: *score* method

The obvious advantage of this method is the reuse of real sessions from the web server log representing real user behavior containing also aborted and resumed sessions. Furthermore, the separated $source$ and $destination$ time frames allow explicit control from which kind of load situations the sessions are selected (scored). This is especially useful when an ISP wants to replay the typical sessions from multiple high load situations, for example.

The selection and rating of the typical sessions results in minor modifications to the properties of the resulting modified workload compared to the original log. The mean values of the metrics will stay while the composition of the sessions is shifted. Untypical sessions are rated lower and thus their number decreases as they will be added later. Also a prediction on the number of added requests is hardly possible as it depends on the original workload, the time frames and the chosen $percent$ and $factor$. Especially, the random insertion of new sessions may result in less number of requests than expected as some requests of a session may be outside the $destination$ time frame.

VI. Constraints for Replay

SLB Distribution: Doing SLB has consequences for dynamic web sites which are based on session management for example. Since a load balancer is ordinarily stateless, it is not guaranteed that requests from the same client are always distributed to the same web server and sessions are probably lost. A benchmark can not avoid this as sessions are generated on the server side. As a solution, user session information could be shared between the backend servers. More sophisticated load balancers like [23] are able to *stick* requests from the same client address to one backend server to solve this issue.

304 Status Code Handling: The `304 Not Modified` status code can result in problems, since a replay benchmark usually does not cache nor does it have access to the cache of the original web clients. Even if the benchmark implements caching there is no guarantee that the resource is available from cache since it is likely that the benchmark replays a specific time frame only and the original resource was requested earlier and hence is not in cache. Hence, within preprocessing of the web server log all 304 status codes should be converted to 200 status codes to persuade the benchmark to re-request the according resources. But this will result in a higher workload.

Idempotent Methods: Another problem is that methods defined as idempotent in the HTTP specification [19] maybe not idempotent. For example, (broken) dynamic web applications may use the `GET` method to modify data. There is no reliable and deterministic way to recognize this automatically without having extra knowledge about the current web application and therefore a replay benchmark can not handle this.

Non-Idempotent Methods: Non idempotent methods will most probably cause errors during the replay. Furthermore, the replay of web server logs does not work for `POST` requests as the key-value `POST` data is not available in web server logs. The analysis of a 107-day period of Wikipedia's traffic [24] has shown a percentage of only 0.02 % requests resulting in a save operation. The same situation was found in the use case (see section VIII). Therefore, *servload* will only support `GET` requests for now.

Timestamp Resolution: A limited timestamp resolution of log entries may result in loosing the arrival order of requests. As this information is lost a benchmark can not correct this.

Web Server Log Information: As already discussed in section IV, the correct identification of user sessions from log entries depends on the log format and the available fields and may fail. Furthermore, a limited proxy server or a load balancer in front of the web server may reduce the available

information in the log also. For example, a proxy or a load balancer can hide or fudge the supported HTTP version of clients from the logs by upgrading or downgrading them on the fly within the session. Similar to the timestamps a benchmark has no chance to work around this.

Maximum Number of TCP Ports: Each user session is bound to TCP connections and according TCP ports. The number of ports is limited to 2^{16} in the TCP header. Furthermore, some port ranges are reserved and should not be used [25]. Thus the number of concurrent sessions from one endpoint to another is limited by the number of free ports.

Maximum Number of Open File Descriptors: Each user session is bound to TCP connections which is associated with file descriptors. The number of open file descriptors is usually limited by the operating system, but can be increased by the system administrator.

Number of Concurrent Connections: Web clients may open multiple concurrent connections to the web server to speedup the fetching of resources. The RFC 2616 [19] recommends two concurrent connections but this is ignored by some web browsers which may open up to eight connections depending on their versions. Different numbers of concurrent connections may result in different benchmark results. It is up to the replay benchmark to provide the same default setting for the number of concurrent connections.

Using only one connection per session is used, all requests are initiated sequential. Obviously, sequential waiting for responses may slow down the request processing at client/benchmark side. This will decrease the workload for the server. To circumvent this HTTP pipelining [19] may be an appreciate technique. Pipelining allows to send multiple ("pipeline") *idempotent* requests without waiting for each response. The web server needs to reply its responses in the same order that the requests were received.

VII. Implementation

Within the implementation the workload generation is separated from the replay. *servprep* is a Lua [26] program which implements the preprocessing, i.e. it is able to identify and analyze sessions and implements the methods *multiply*, *peak*, and *score*. The replaying benchmark *servload* is written in *C*. The preprocessed request data of *servprep* is transformed to a format which can be used by the replaying benchmark *servload* (see Figure 1).

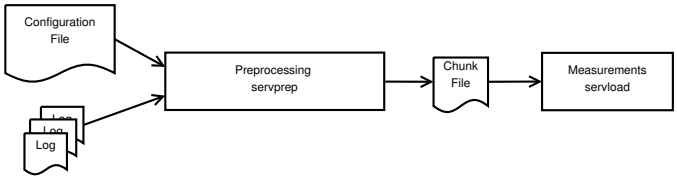

Figure 1. *servprep* generates workload for *servload.*

servprep reads a configuration file with the required parameters on startup. This configuration file contains some generic settings like the target host and target port, and a boolean to determine if Secure Sockets Layer (SSL)/Transport Layer Security (TLS) should be used.

servprep analyzes the generated workload and plots the resulting changes. These steps, workload generation and analysis, can be repeated to create and test several workloads from different parameters.

servload loads requests from the *chunk* file generated by *servprep* and replays them. Support is given for a subset of features of HTTP version 1.1 for example `GET`, `HEAD` and `OPTIONS` requests over persistent connections on IPv4 and IPv6. Furthermore rudimentary SSL/TLS support, concurrent user sessions and user agent header fields are implemented.

servload uses only one connection per session and thus is affected by the serialization limits for each session described in section VI. As countermeasure, it implements HTTP pipelining. *servload* does not implement any caching mechanism to handle 304 status codes as mentioned in VI. Instead, it re-requests all resources as *servprep* converts all 304 status codes to 200 status codes in the previous analysis. A warm-up and tear-down phase is not yet implemented but can be simulated by running a workload three times. First time to warm-up the system, second time to measure, and third time to tear down.

The program flow is based on `poll()` events. After parsing the *chunk* file and initializing the required structures, the replay of the requests is started in a loop until all requests are done. Possible think times are respected within the loop and all currently active and non-blocking connections are checked with `poll()` to handle read, write or error events. After all requests are processed the results are analyzed and the according metrics are printed to the standard out. This output includes the general input settings, the total number of requests, connections and transferred bytes, the duration and the resulting connection and data throughput as well as the connect time, the response time and all connection related errors. Furthermore, the protocol specific metrics are put to standard out as well like the response timeouts together with SLA value and the number of response errors resulting from incomplete requests for example.

VIII. Evaluation

The performance tests for the *servprep* and *servload* implementation are done in the production environment of an ISP.

The tests evaluate the presented workload generation methods *multiply*, *peak*, and *score*. Further, the replay functionality of *servload* is tested. Finally, measurements in an SLB environment of the ISP are shown to prove the suitability of *servload* for such environments.

A. ISP Use Case

The ISP use case is to provide web sites, web based services and applications for mobile devices. Web sites are rendered for each device to make the best use of the limited display capabilities of the clients. The dynamic web site generation is resource intensive for the web servers and is therefore done on a web server cluster which even can handle only a few

requests per second. The ISP provided a web server log of an advertising campaign.

Log entries can be customized and extended by additional fields. In the ISP use case additional fields are appended. Among others the *session id*, a generated id to identify user sessions, is added. This session id field is considered by *servprep* and reveals unique user sessions from web server log files.

B. Measurement Environment

The ISP testbed consists of a web client running *servload* located in Europe/Berlin and a web server cluster located in South Korea. Therefore, besides the hardware and the software configuration the latency and the according throughput between the web client and the web server have impact on the measurement results.

The web client is an 1.8 GHz AMD Sempron 3100+ with 1 GByte RAM running Debian Linux with Kernel 2.6.25-2-486, Lua 5.1 and GCC 4.1.2 installed and connected with a 10 MBit uplink. The web server cluster consists of four nodes and each is an 2 CPU 2.0 GHz Xeon E5504 (4 Cores) with 16 GByte RAM running Red Hat Enterprise Linux with Kernel 2.6.18-128.el5, Apache HTTP server 1.3.41 with mod_perl 1.30, MySQL 5.0.77 and Perlbal 1.73 installed connected with a 100 MBit uplink.

Measurements are done in a *single web server setup* with one client node and one web server node as well as in a *web server cluster setup* with one client node, a load balancer node (running Perlbal) and two web servers nodes behind the load balancer. The chosen time frame from the original web server log is 30 minutes long and contains the highest number of request per second found in the whole log.

Additionally to the metrics provided by *servload*, the CPU usage and the number of incoming requests is monitored on the web cluster by Munin [27].

C. Generation of Workload

First, the *score* method is tested which adds *representative* sessions. In the experiment, we chose a $factor$ 1 and 0.5 modification, which results in 50 % more requests. The experimental results are shown in Table I.

The session metrics median number of requests, median length, and median bytes of the original and modified log have similar mean and average values. The standard deviation has to get smaller, since we add/replicate a huge amount of sessions. The mean numbers are always in the expected range. Compare for example the 11 requests per session in the original trace with 13 requests per session in the generated trace. The average also has to get smaller, since score does not add outliers.

This confirms our approach to add whole (original) sessions.

D. Replay

For the evaluation of the replay functionality of *servload*, the metrics of the original log are compared with the web server log of the replay. The original web server log was taken from the *web server cluster setup* while the replay was done within the *single web server setup*.

Metric	Original	Modified
Requests	102081	153118
Sessions	5684	8656
Length (s)	86375	86375
Requests per second		
Minimum	0	0
Maximum	15	18
Median	1	1
Average	1.18	1.52
Standard deviation	1.52	1.77
Number of Request		
Minimum	1	1
Maximum	385	385
Median	11	13
Average	17.96	17.69
Standard deviation	21.57	18.88
Median Think-Time (s)		
Minimum	0	0
Maximum	60768	60768
Median	3	2.5
Average	514.71	321.54
Standard deviation	3431.11	2718.14
Length (s)		
Minimum	0	0
Maximum	85851	85851
Median	90	96
Average	2578.67	1776.90
Standard deviation	11536.65	9500.495
Median Bytes		
Minimum	0	0
Maximum	73890	73890
Median	1275	1319
Average	2205.78	2009.35
Standard deviation	3277.69	2740.71

Table I
ANALYSIS OF ORIGINAL AND MODIFIED WEB SERVER LOG.

The results are shown in Table II. All requests were processed correctly with the expected status codes and therefore the error metrics are left out as they were all zero. The median number of requests and the median think time fits perfectly. The total length (1800 versus 1809) and the session length (166.5 versus 168) differ slightly. This is not astonishing since the experiment was done in the company testbed with a non-dedicated network between Europe and South Korea. Since the values are increased in the replay case, the effect may also be caused by the disabled (as unfinished) HTTP pipelining implementation in *servload* resulting in sequentially processed requests within a single session. This explains also the decrease of the maximum requests per second (from 27 to 16). There is also a slight difference in the bytes metric (1245 versus 1314). This may result from requests with 304 status codes which are not cached by *servload* and time dependent generation of output.

All together, the metrics match pretty well which suggests that *servload* has done the replay correctly.

IX. Conclusion and Future Work

Web server applications belong to the most important Internet applications and scalability tests are of special interests for

Metric	Original	Replay
Requests	10024	10024
Sessions	118	118
Length (s)	1800	1809
Requests per second		
Minimum	0	0
Maximum	27	16
Median	5	5
Average	5.57	5.54
Standard deviation	4.37	2.62
Number of Request		
Minimum	1	1
Maximum	663	663
Median	52	52
Average	84.95	84.95
Standard deviation	102.71	102.71
Median Think-Time (s)		
Minimum	0	1
Maximum	1211	1211
Median	0	1
Average	39.20	39.78
Standard deviation	172.18	172.08
Length (s)		
Minimum	0	0
Maximum	1798	1798
Median	166.5	168
Average	414.85	416.86
Standard deviation	526.36	526.06
Median Bytes		
Minimum	287	288
Maximum	12394	12395
Median	1245	1314
Average	2024.22	2037.55
Standard deviation	1921.26	1893.88

Table II
ANALYZE OF ORIGINAL AND REPLAY WEB SERVER LOG.

ISPs to estimate performance numbers for SLAs contracted to customers. Therefore, support to generate *representative* higher workloads is necessary.

This paper presents the design and implementation of *servload*, a tool to generate and replay *representative* workloads from web server logs. Given a real web server log, *servload* uses the so-called score method to identify and add *typical* user sessions. By adding complete user sessions, *servload* generates a representative higher workloads.

The analysis functionality differs from conventional log analyzers like AWStats [28] as these tools have no need for modifying the requests later.

The functionality of *servload* is proven in a real production environment. A replay of the original workload was very close to the original one. Furthermore, in opposite to benchmarks like *SPECweb2009*, *servload* can be used in load balancing scenarios as well.

Currently, *servload* is extended to support further protocols like DNS.

REFERENCES

[1] J. Lehmann, L. Schneidenbach, B. Schnor, and J. Zinke, "Self-Adapting Credit Based Server Load Balancing," in *"Proceedings of the IASTED international Conference on Parallel and Distributed Computing and Networks (PDCN)"*, ser. PDCN, H. Burkhart, Ed., vol. 0, "IASTED". Innsbruck, Austria: "ACTA Press", February 2008, pp. 55–62, "ISBN: 978-0-88986-713-0, ISBN (CD): 978-0-88986-714-7".

[2] J. Zinke, "Self adapting load balancing network," http://www.salbnet.org/, accessed 11/09.

[3] W. Zhang, "The Linux Virtual Server Project - Linux Server Cluster for Load Balancing," http://www.linuxvirtualserver.org/, accessed 11/11.

[4] S. P. E. Corporation, "SPECweb2009," http://www.spec.org/web2009/, accessed 11/08.

[5] A. L. Webmaster, "http_load," http://www.acme.com/software/http_load/, accessed 11/08.

[6] J. D. Software, "Siege Home," http://www.joedog.org/index/siege-home, accessed 11/08.

[7] C. Goldberg, "Pylot - Open Source Web Performance Tool," http://www.pylot.org/, accessed 11/08.

[8] H.-P. D. C. L.P., "Welcome to the httperf homepage," http://www.hpl.hp.com/research/linux/httperf/, accessed 11/08.

[9] N. Niclausse, "Tsung," http://tsung.erlang-projects.org/, accessed 11/08.

[10] P. Barford and M. Crovella, "Generating representative web workloads for network and server performance evaluation," in *Proceedings of the 1998 ACM SIGMETRICS joint international conference on Measurement and modeling of computer systems*, ser. SIGMETRICS '98/PERFORMANCE '98. New York, NY, USA: ACM, 1998, pp. 151–160.

[11] A. Savoia, "The science of web site load testing," http://www.keynote.com/docs/whitepapers/science_of_web_load_testing.pdf, Keynote Systems, Inc., 2000, accessed 11/08.

[12] L. Xu, W. Zhang, and L. Chen, "Modeling users' visiting behaviors for web load testing by continuous time markov chain," in *Proceedings of the 2010 Seventh Web Information Systems and Applications Conference*, ser. WISA '10, vol. 0. Washington, DC, USA: IEEE Computer Society, 2010, pp. 59–64.

[13] J. Sant, A. Souter, and L. Greenwald, "An exploration of statistical models for automated test case generation," in *Proceedings of the third international workshop on Dynamic analysis*, ser. WODA '05, vol. 30. New York, NY, USA: ACM, 2005, pp. 1–7.

[14] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber, "Realistic load testing of web applications," in *Proceedings of the Conference on Software Maintenance and Reengineering (CSMR)*, vol. 0. Washington, DC, USA: IEEE Computer Society, 2006, pp. 57–70.

[15] J. Grundy, "MaramaMTE," https://wiki.auckland.ac.nz/display/csidst/MaramaMTE, accessed 11/08.

[16] X. Wang, B. Zhou, and W. Li, "Model based load testing of web applications," in *Proceedings of the International Symposium on Parallel and Distributed Processing with Applications*, ser. ISPA '10, vol. 0. Washington, DC, USA: IEEE Computer Society, 2010, pp. 483–490.

[17] M. S. Johns, "Identification protocol," Internet Engineering Task Force, RFC 1413, February 1993. [Online]. Available: http://www.rfc-editor.org/rfc/rfc1413.txt

[18] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. J. Leach, A. Luotonen, and L. Stewart, "HTTP authentication: Basic and digest access authentication," Internet Engineering Task Force, RFC 2617, June 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2617.txt

[19] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. J. Leach, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1," Internet Engineering Task Force, RFC 2616, June 1999. [Online]. Available: http://www.rfc-editor.org/rfc/rfc2616.txt

[20] N. Ltd, "Netcraft - January 2011 Web Server Survey," http://news.netcraft.com/archives/2011/01/12/january-2011-web-server-survey-4.html, accessed 11/08.

[21] T. A. S. Foundation, "The Apache HTTP Server Project," http://httpd.apache.org/, accessed 11/08.

[22] D. A. Menascé, "Load testing of web sites," *IEEE Internet Computing*, vol. 6, pp. 70–74, July 2002.

[23] OpenBSD, "PF: The OpenBSD Packet Filter," http://openbsd.org/faq/pf/index.html, accessed 11/11.

[24] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Elsevier Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009.

[25] I. A. N. Authority, "PORT NUMBERS," http://www.iana.org/assignments/port-numbers, accessed 11/08.

[26] PUC-Rio, "The Programming Language Lua," http://www.lua.org/, accessed 11/08.

[27] J. Olsen, "Munin," http://munin.projects.linpro.no/, accessed 11/09.

[28] L. Destailleur, "AWStats - Free advanced log file analyzer for web, ftp or mail statistics (GNU GPL)." http://awstats.sourceforge.net/, accessed 11/08.