View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Dagstuhl Research Online Publication Server

Web Workload Generation According to the UniLoG Approach

Andrey W. Kolesnikov and Bernd E. Wolfinger

University of Hamburg, Department of Informatics
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
{kolesnikov,wolfinger}@informatik.uni-hamburg.de

Abstract

Generating synthetic loads which are sufficiently close to reality represents an important and challenging task in performance and quality-of-service (QoS) evaluations of computer networks and distributed systems. Here, the load to be generated represents sequences of requests at a well-defined service interface within a network node. The paper presents a tool (`UniLoG.HTTP`) which can be used in a flexible manner to generate realistic and representative server and network loads, in terms of access requests to Web servers as well as creation of typical Web traffic within a communication network. The paper describes the architecture of this load generator, the critical design decisions and solution approaches which allowed us to obtain the desired flexibility.

1998 ACM Subject Classification C.2.2 [Computer Communication Networks]: Network Protocols – HTTP; C.2.3 [Computer Communication Networks]: Network Operations – Network management; C.2.4 [Computer Communication Networks]: Distributed Systems – Distributed applications; C.4 [Performance of Systems]: Measurement techniques, Modeling techniques, Performance attributes; G.3 [Probability and Statistics]: Distribution functions, Experimental design, Stochastic processes; I.6.5 [Simulation and Modeling]: Model Development – Modeling methodologies.

Keywords and phrases Web workload generation, Web traffic generation, Unified Load Generator, Performance Evaluation, HTTP/1.1

Digital Object Identifier 10.4230/OASIcs.KiVS.2011.49

1 Introduction

Generating load for computer and communication networks in a sufficiently realistic manner represents an important and challenging task. Realistic load generation typically is an indispensable prerequisite to the analysis and prediction of network performance and of quality-of-service (QoS). Moreover, load generation may support functionality testing of network components (on various levels of component utilization) or the testing of the effectiveness of security mechanisms.

Load generation can be required at very different interfaces within a computer network, such as user-/application-oriented interfaces at which sequences of requests have to be generated like requests to a Web server, files to be transferred, sequences of video frames or of digitized voice data to be transmitted, etc. On the other hand, it might also be necessary to generate load for a lower-layer interface within a protocol hierarchy, such as creation of IP packets or of Ethernet frames to be transmitted.

An interesting approach for load generating, from our point of view, is the provisioning of a unified tool which allows one to generate load (i.e. sequences of requests) at an interface which can be chosen by an experimenter dependent on the kind of study he/she is carrying out. This approach has been followed by the authors during the development of the unified

© Andrey W. Kolesnikov and Bernd E. Wolfinger;
licensed under Creative Commons License NC-ND
17th GI/ITG Conference on Communication in Distributed Systems (KiVS'11).
Editors: Norbert Luttenberger, Hagen Peters; pp. 49–60

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

load generator `UniLoG` [1]. The basic principle underlying the design and elaboration of `UniLoG` has been to start with a formal description of an abstract load model and thereafter to use an interface-dependent adapter to map the abstract requests to the concrete requests as they are "understood" by the service providing component at the real interface in question. Our earlier research covers the provisioning of adapters for TCP and UDP [2] as well as IP [1] interfaces. Because of the pre-eminent importance of the World Wide Web (WWW) a strong desire emerged to embed the capability into `UniLoG` of generating realistic Web traffic and realistic Web server loads, too.

The following strongly different modeling approaches (MA) are possible and may be useful for various kinds of studies when characterizing HTTP loads and the resulting Web traffic induced by this load:

(MA_1) Realistic description of sequences of requests as they are passed to the HTTP interface $IF_c(HTTP)$ within the Web client. Generating a (realistic) sequence of requests at interface $IF_c(HTTP)$ allows one, with only little expenditure, on the one hand to generate some specific load for a Web server (in terms of client requests of varying complexity) and on the other hand to generate some specific load for the network (in terms of Web traffic of different structure and intensity) [3, 4].

(MA_2) Realistic description of sequences of requests as they are passed, both, in the web client C (as client requests) and in the Web server S (as server responses) to the corresponding HTTP interfaces $IF_c(HTTP)$ and $IF_s(HTTP)$. As server responses are generated as a consequence of client requests, precise modeling here requires coordination of load generation at both interfaces – and this is a tedious task. The concurrent generation of sequences of requests at interfaces $IF_c(HTTP)$ and $IF_s(HTTP)$ replaces the Web server being used in MA_1 by a load generator [5, 6]. Therefore, generation of load here gets much more complicated because a sufficiently realistic model for the Web server is required and load generation in the client and the server have to be coordinated. Moreover, this approach does no longer allow the dedicated loading of a Web server with client requests.

(MA_3) Realistic description of the sequence of IP packets (as observed at the IP interface $IF_s(IP)$ in the Web server) or requests at the TCP service interface ($IF_s(TCP)$ in the server) as they are induced by single or an overlay of client requests to the server. The IP packets or TCP requests result from the transmission of the corresponding server responses. This modeling approach fundamentally differs from MA_1 and MA_2 as it assumes a completely different interface for load description/generation (IP instead of HTTP interface). The approach is useful in cases, when, e.g., streams of IP packets have to be injected into a network in a manner as they would have been induced by single or an overlay of Web server accesses. Most of the currently existing literature on characterization/generation of HTTP loads [7, 8] follows this modeling approach, though, in order to be precise, one would have to consider it as a characterization/generation of IP packet streams. Here also, the dedicated loading of a Web server with client requests is impossible using this approach.

MA_1 is the approach followed by us and presented in this paper, because we want to generate load for the server, too, and not only for the network (which is impossible when using approaches MA_2 or MA_3). Moreover, we want to inject the load directly at the HTTP interface and therefore MA_3 is again not an alternative to the MA_1 approach.

A survey on the literature related with this work follows in Sec. 2. The application of our unified approach to Web workload generation is illustrated in Sec. 3. In Sec. 4, the architecture of the corresponding HTTP adapter along with an algorithm for the allocation

of real requests from a pool of Web sites are explained. The estimation of the relevant workload/traffic characteristics and the construction of a sufficiently large, representative and stable pool of Web sites for load generation are presented in Sec. 5 and Sec. 6. A short summary and an outlook on future work conclude this paper.

2 Related Work

Web workload generators are software tools based either on traces reflecting real Web user sessions or on workload models that are designed and implemented to generate HTTP requests. Floyd and Paxson demonstrated in their study [9] how difficult it is to generate representative Web requests, especially when some particular characteristics in a dynamic Web site should be modeled, and how these characteristics impact on the behaviour of the Web clients.

One of the first studies trying to identify the common characteristics in Web server workloads is the work done by Arlitt and Williamson [10], which used logs of Web server accesses at six different sites (three from university environments, two from scientific research organizations, and one from a commercial Internet service provider). The observed workload characteristics were used to identify the possible strategies for the design of a caching system to improve Web server performance.

Barford and Crovella [4] applied a number of observations of Web server usage to create a realistic Web workload generation tool, called `SURGE` (Scalable URL Reference Generator) which mimics a set of real users accessing a server and generates URL references matching empirical measurements of request and server file size distribution, relative file popularity, embedded file references, temporal locality of reference, and idle periods of individual users. The relevance of these Web workload characteristics as well as their concrete values were identified based on single (nonrecurring) measurements, so that later revisiting done by Williams et al. [11] was required due to emerging Web technologies and a 30-fold increase in overall traffic volume in 2005.

Rolland et al. [8] proposed the open-loop packet-level traffic generator `LiTGen`, which statistically models IP traffic (resulting from Web requests) on a per user and application basis and, in contrary to the recommendations of Paxon and Floyd [9], does not consider such important network and protocol characteristics like RTT, link capacities or TCP dynamics.

`ParaSynTG` [6] is a synthetic trace generator for source-level representation of Web traffic with different characteristics such as document size and type, popularity (in terms of frequency of reference) as well as temporal locality of requests. The tool was designed for the generation of workload traces only (which can be used, e.g., in simulation experiments) and, in the opposite to the design objectives of our Web workload generator `UniLoG`, provides no facilities to generate and to inject real HTTP requests into the network.

`PackMime-NS` [5] is an example for a source-level and `Swing` [7] is an example for a packet-level traffic generation tool, respectively, with a lot of effort spent by the authors on the ability to take into account the network and protocol characteristics (e.g., RTT, link capacities and error rates, dynamic TCP interactions between Web clients and servers). As a consequence, there are not only the Web clients and Web servers that have to be modeled by these solutions, but also the other components of the network under study. Hence, the field of application of these solutions is restricted to network simulation experiments (whereas our solution is targeting performance evaluation studies for real networks and Web servers, too).

A further survey of traffic generation tools (including the solutions for other interfaces, e.g., Ethernet, IP, TCP/UDP) can be found in [12].

3 Application of the UniLoG Approach to Web Traffic Generation

The characteristic behaviour of a user navigating Web sites by means of a browser is presented in Fig. 1 (left). A typical Web site consists of a base object and multiple embedded objects. With the first HTTP request the browser retrieves the base object of the site, parses it and issues subsequent HTTP requests to fetch all embedded objects.

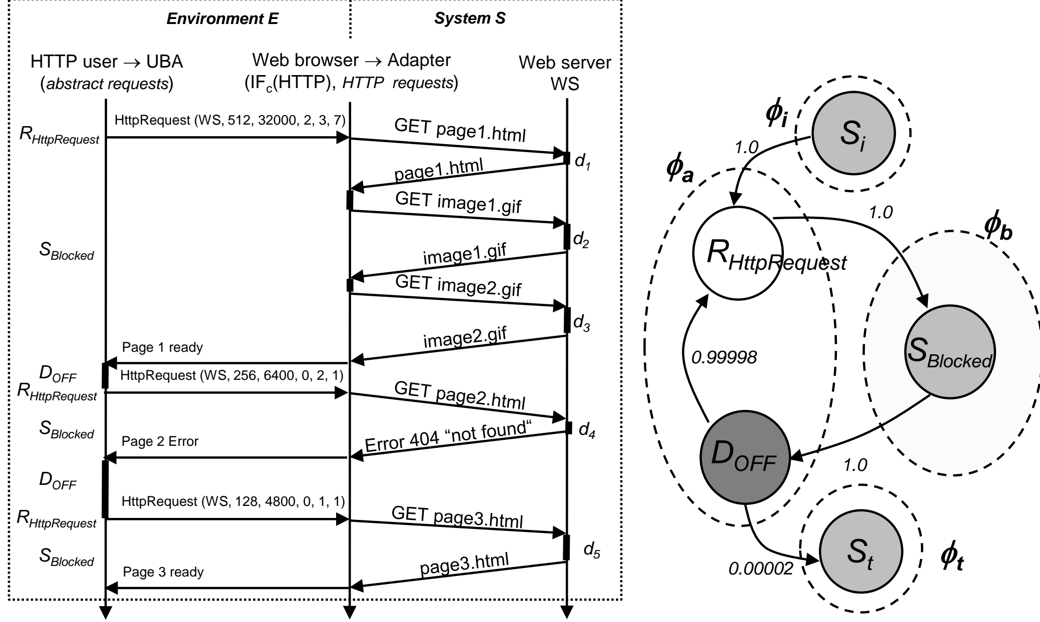

Figure 1 Retrieval of multiple pages and embedded objects from the server WS (left) and the corresponding UBA (right).

According to the unified load generation approach presented in [1], a set of HTTP service users which are relevant for the particular modeling task at the interface $IF = IF_c(HTTP)$ in the Web client is considered as an environment E while layers of the protocol stack below the HTTP interface in the Web client, the Web server, and the communication network are considered as being part of the service system S. The workload $L = L(E, S, IF, T)$ is furthermore represented by a sequence of requests $(t_i, r_i), t_i \in T, i = 1, 2, ..., n$ ($t_i \in \mathbb{R}$: arrival time of request r_i at IF, $t_i \leq t_j$ for $i < j$, $n \in \mathbb{N}$) passed by the environment E to the service system S at the interface $IF = IF_c(HTTP)$ during a time interval T.

Each request r_i is characterized by the corresponding abstract request type which is chosen from a set $\mathbb{RT} = \{RT_1, RT_2, \ldots, RT_m\}, m \in \mathbb{N}$, of abstract request types supplied by the experimenter. There is nearly a dozen of different request methods provided by the HTTP/1.1 protocol specification (cf. RFC 2616) for the interaction with the Web server. For the specification of an HTTP workload model, the experimenter can decide to provide different abstract request types for each HTTP request method but he/she would often prefer to define only one single abstract request type (e.g., `HttpRequest`, cf. Fig. 1) addressing HTTP requests in general, and to concentrate on the identification of request parameters which have a significant impact on the workload induced in the server and network. The analysis of the workload characteristics used in a series of studies on Web workload modeling (e.g., [3, 4, 5, 11]) drove the choice of the following request parameters to be provided with our solution by default.

serverName contains either the fully qualified domain name (FQDN) or the IP address of the target Web server and has a direct influence on the generated network traffic matrix. In case the experimenter plans to test a particular Web server or a specific function of a Web service deployed onto it, the exact location of the object to be demanded can be supplied by this parameter (e.g., by means of a URL including a full path and search part, if needed).

replySize is the total amount of response data from the server including the size of all embedded objects. This attribute is not part of the HTTP/1.1 message but it has a significant impact on the load induced by HTTP request.

numberOfEmbeddedObjects specifies the number of embedded objects (e.g. images, links, script objects) in the requested page. HTTP/1.1 allows a single TCP connection to be used for multiple object requests.

complexity is provided to characterize the structural complexity of the page in general and can be specified by means of a complexity class derived from the values of page complexity characteristics `numberOfEmbeddedObjects` and `replySize` which do not consider the induced server load directly.

requestSize is the total amount of data transferred from the client to the Web server, which is affected by the number and the size of requests for the embedded objects in the page. Some of HTTP request headers (e.g. `Accept`, `User-Agent`) may also influence the size of HTTP messages [13].

inducedServerLoad characterizes the amount of time required for the server to respond to the client request and can be specified by means of a server delay class derived from values of particular server delays ($d_1 - d_5$, cf. Fig. 1) induced by requests to the main page and its embedded objects. Client requests may induce further searching, authentication or database retrieval activities on the server side and usually differ significantly in this factor.

The set $\mathbb{RT}$ of abstract request types is constructed by including the relevant request parameters from the list above into the corresponding request types $RT_r = Id(RT_r) \cup \{a_1, a_2, \ldots, a_p\}, RT_r \in \mathbb{RT}, r \in \mathbb{N}, 1 \leq r \leq m$, as abstract request attributes $a_k, k \in \mathbb{N}, 1 \leq k \leq p$, where $p \in \mathbb{N}, p = p(r)$ is the number of attributes in RT_r and $Id(RT_r)$ is its unique name. In this way, the experimenter can define abstract request types with different level of abstraction dependent on the kind of study being carried out. For example, the experimenter can decide to include only one single attribute `inducedServerLoad` into the abstract request type `HttpRequest` in order to analyse the utilization level of some known Web server. In case he/she plans to test a specific function of a Web service/application deployed to that server, the attribute `serverName` containing the full URL of the referenced object is most likely to be added to `HttpRequest`. In order to produce various background loads for the network in terms of Web traffic of different structure and intensity, further attributes like `numberOfEmbeddedObjects` or `replySize` can be included into `HttpRequest`.

For the specification of the possible sequences of requests, the set of relevant HTTP service users is represented by a user behaviour automaton (UBA, cf. Fig. 1). A UBA is an extended finite automaton $U = \{\phi, T_\phi\}$ consisting of the set of macro states $\phi = \{\phi_i, \phi_a, \phi_b, \phi_t\}$ which describe the four typical types of user activity (initialization in ϕ_i, generation of requests in the active macro state ϕ_a, waiting for system reactions in the blocked macro state ϕ_b, termination in ϕ_t) and the set T_ϕ of transitions between these macro states. The macro states are further refined by means of (R)equest-, (D)elay- and (S)ystem-states, which are introduced to model the generation of requests of a particular abstract request type, the

delay times between successive requests and/or system events, and waiting for a certain type of system event to occur, correspondingly (cf. [1]).

Consider a simple UBA for an HTTP user as presented in Fig. 1. The user starts in the S-state S_i of the initialization macro state ϕ_i of the UBA and, after the Internet browser is started completely, becomes active by changing its macro state to ϕ_a. To model the retrieval of Web pages by the user, requests of abstract request type `HttpRequest(serverName, requestSize, replySize, numberOfEmbeddedObjects, inducedServerLoad, complexity)` are generated in the R-state $R_{HttpRequest}$ of ϕ_a. The values of request attributes can be specified by the experimenter by means of constant values, traces of real measurements or various distribution functions. We note that these requests are initially abstract, i.e. they contain attributes (e.g., `replySize`, `inducedServerLoad`) which differ from the attributes of real HTTP messages significantly or do even not exist in HTTP messages at all. Moreover, one `HttpRequest` usually induces more than one real HTTP request to retrieve images, links and script objects embedded in the page as well as objects linked from other servers (e.g., advertisement pop-ups and Web bugs). For example, during the execution of the UBA, an abstract `HttpRequest(WS, -1, 32000, 2, -1, -1)` represents an instruction at our load generator to invoke a Web page on the server `WS` which contains 2 embedded objects resulting in the total reply size of 32000 byte (a value of -1 means that the corresponding parameter is not used in this request). So, the `UniLoG.HTTP` adapter must issue the corresponding real HTTP requests for the main object and for each of the two embedded objects of the page (cf. Fig. 1).

In the simple UBA presented here, the user resides in the S-state $S_{Blocked}$ of the blocked macro state ϕ_b until the page is retrieved completely (i.e. until the last embedded object is fetched successfully or an error/status code is returned). There are, in general, other possibilities to model this behaviour (e.g., to leave $S_{Blocked}$ immediately after the main object of the page is loaded). After a user-dependent delay time ("think time"), modeled in the D-state D_{OFF}, the next page is accessed by the user (by following a link on the current page or by entering a URL in the address line of the browser directly) and the corresponding `HttpRequest` is generated in $R_{HttpRequest}$. The UBA is executed until the S-state S_t of the termination macro state ϕ_t is reached or the upper limit of the time interval T specified by the experimenter for load generation is exceeded.

It should be noted, that the experimenter can specify advanced Web workload characteristics by means of a UBA, which are not explicitly provided by the set of abstract request parameters. For example, the user think time (which is an important property of Web traffic to capture its bursty nature [4]) can be specified in D-states as interarrival time between subsequent requests `HttpRequest`. Page popularity (defined as relative number of requests made to individual pages on the server) can be specified, e.g., by means of a frequency or Zipf-like distribution for the `serverName` parameter (in this case, containing the full URL of the page to be referenced) of `HttpRequest`. Temporal locality of page requests (referring the likelihood that, once a page has been requested, it will be requested again in the near future) can be characterized by means of a distribution of stack distances [6] for the `serverName` parameter (again, containing the full URL of the page to be referenced) of `HttpRequest` in the request sequence to be generated by our load generator.

4 Allocation of Real HTTP Requests

The Unified Load Generator `UniLoG`, as presented in [1], incorporates a formal automata-based load specification technique and exhibits a distributed architecture to provide a high

degree of flexibility in generating traffic mixes of various structure and intensity for different scenarios. The experimenter can define workload models in form of a UBA, and distribute the parameterized UBA (PUBA) to the load generating nodes (load agents) involved in the experiment, which can be controlled by the experimenter from one central point in the network (management station, cf. Fig. 2). In each load agent, the generator component (GAR) for the PUBA execution and one or many adapters (e.g. `UniLoG.HTTP`) for the allocation of the corresponding real requests are installed.

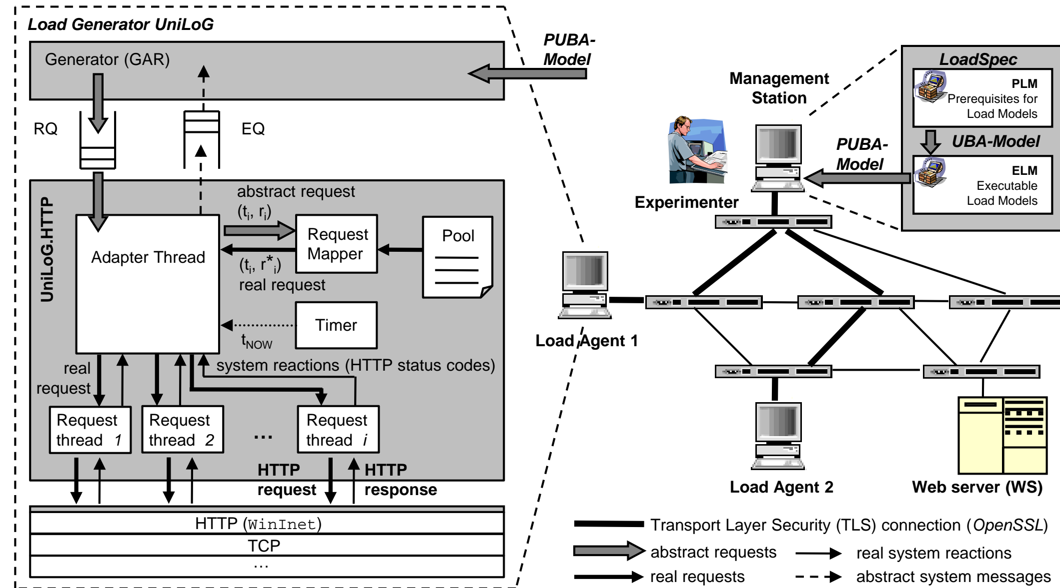

Figure 2 `UniLoG` architecture and the main components of the `UniLoG.HTTP` adapter (left).

The key components of the `UniLoG.HTTP` adapter are presented in the left part of Fig. 2. The main adapter thread consumes abstract requests (t_i, r_i) which are generated by the generator thread (GAR) in R-states of the current PUBA and inserted into the request queue RQ. For each abstract request the adapter invokes the request mapper which is responsible for the allocation of the correspondent real HTTP request from the pool. Each HTTP request in the pool is represented by a tuple $e_i = (v_{i,1}, v_{i,2}, \ldots, v_{i,p}, v^*_{i,1}, v^*_{i,2}, \ldots, v^*_{i,r}), i \in \mathbb{N}, 1 \leq i \leq N$. In such a tuple, $v^*_{i,1}, \ldots, v^*_{i,r}$ denote the values of r real request attributes required to build a request which is conform to HTTP/1.1 (like HTTP request method `requestMethod`, server name `serverName` and port `serverPort`, object name `objectName`, and the optional data `optData` of length `optLength` sent with POST requests). $v_{i,1}, \ldots, v_{i,p}$ denote the corresponding values of p abstract request parameters (defined in Sec. 3), which can be measured manually by the experimenter or estimated by the adapter automatically (see Sec. 5) and used to query the pool for real requests which match as good as possible the current abstract request. Finally, N denotes the pool size, i.e. the number of its elements.

To allocate the best matching real request (t_i, r^*_i), the request mapper starts with the abstract parameter a_1 of the highest priority (the priority of the abstract request parameters can be specified by the experimenter in PUBA) and creates a candidates list consisting of requests from the pool which exhibit the minimal distance between a_1 and $v_{i,1}, 1 \leq i \leq N$. In the next step, the abstract parameter a_2 with the second highest priority is selected and only the requests with the minimal distance between a_2 and $v_{i,2}, 1 \leq i \leq N$ are kept on the candidates list. In case of string parameters, especially for `serverName`, an exact match in

place of the minimal distance candidates is required. This procedure is repeated until only one candidate is left or all of the p abstract request parameters are processed. In case there are more than one real requests left on the candidates list after the last step, one of them is selected randomly.

The request allocation algorithm is flexible in the number and types of abstract request parameters used for the pool query. Its complexity is determined by the number v of required comparisons which yields to $v = N \cdot p$ for the worst case when all N real requests from the pool match each of the p parameters of the current abstract request.

The allocated real request (t_i, r_i^*) is prepared for execution by creating a new request thread and establishing the connection to the Web server, if needed. Thereafter, the adapter thread waits until the specified request injection time t_i is reached and resumes the request thread for execution. HTTP status codes returned by the request thread represent the system messages which are inserted by the adapter thread into the event queue EQ.

As a proof of the presented concept, the first prototype of the `UniLoG.HTTP` adapter for Windows was developed in [14]. Meanwhile, this prototype has been significantly improved to imitate the behaviour of a real browser more closely and, finally, it has been integrated in the `UniLoG` load generator. `UniLoG.HTTP` makes use of the `WinInet` library for the management of Web server connections as well as for the formation and injection of HTTP requests into the network. The choice of `WinInet` is motivated by the fact that it is internally used by the Internet Explorer (IE) browser itself and thus enables the adapter to imitate the behaviour of this browser in a very realistic manner.

For this reason, we applied the same values for the `User-Agent` and `Accept` request header fields as they are used by the IE 7.0 [1] during our `HttpOpenRequest` calls for the formation and injection of HTTP requests in the adapter. Furthermore, similar to [13], a series of `INTERNET_FLAGs` (`NO_CACHE_WRITE`, `RELOAD`, `PRAGMA_NO_CACHE`, `KEEP_CONNECTION`) has been specified in `HttpOpenRequest` calls in order to instruct the adapter to retrieve the requested objects from the original server and not from the local cache or proxy, using, if possible, a keep-alive connection. These measure should help to ensure that the adapter induces Web traffic with characteristics as specified by the values of abstract request parameters in the pool entries.

5 Estimation of Values for Abstract Request Parameters

It becomes apparent, that a sufficiently large pool of Web requests with properly estimated values of their abstract parameters is required to take `UniLoG.HTTP` in operation. Furthermore, the pool should be kept up-to-date to ensure that the characteristics of the Web traffic induced by `UniLoG.HTTP` correspond to the values of abstract parameters specified in the pool entries.

In the measurement approach applied in [15], a series of Firefox browser sessions were traced with `tcpdump` and analyzed to extract various characteristics of worldwide top 500 popular web sites. In this way, all components of the requested page are considered, providing very realistic Web server workload and traffic characteristics. However, this method leads to a sufficient programming effort for combining a series of external tools to fully automate the measurements, whereas the required measurement points can be accessed by us in the `UniLoG.HTTP` source code directly. Furthermore, `UniLoG.HTTP` is aimed at imitating the

[1] These values were extracted from the registry key `HKLM/Software/Microsoft/Windows/CurrentVersion/InternetSettings`

behaviour of the real browser to some extent and is not able yet to interpret all existing types of embedded objects (e.g. JavaScript or CSS objects are loaded but not interpreted). Therefore, we decided to integrate the parameter estimation function for our pool immediately into the adapter and allowed it to be turned on and off by the experimenter. The values of abstract parameters for each pool entry are estimated during the retrieval of the correspondent Web page and its embedded objects from the server according to the following rules.

numberOfEmbeddedObjects (o) the adapter retrieves the main object of the requested page specified by the concatenation of abstract `serverName` and real `objectName` parameters and parses its content to recognize the embedded images, links, and script objects to retrieve them by means of consequent requests from the same server. Objects linked from other servers are not loaded and hence not considered in o at this moment.

replySize (s) is computed as the total amount of data (in kbyte) loaded by the adapter during the `InternetReadFile` calls for the main page and all its embedded objects.

complexity class $C \in \{C_1, \ldots, C_9\}$ for the page is derived from its estimated values o and s using class boundaries o_1, o_2 and s_1, s_2 which can be supplied by the experimenter for the `numberOfEmbeddedObjects` and `replySize` parameters, correspondingly (cf. Fig. 3).

requestSize is calculated as the total amount of data (in byte) sent by the adapter to the server considering requests for the main page, all its embedded objects and the optional data for POST requests. The mean size of GET requests issued by the IE 7.0 browser was estimated to 612 byte in [16].

inducedServerLoad class $D \in$ {`Immediately`, `Fast`, `Slow`, `Annoyingly slow`} is derived from the delay times $d_i, i = 1, \ldots, o$, induced on the server by requests to the embedded objects of the page. The delay time induced by the main object is not considered to exclude the time spent for potential requests to the DNS server. Each d_i is estimated as a difference between request execution time a_i (cf. Fig. 3) and the server round trip time (RTT_s), estimated either from the TCP handshake in the beginning of the HTTP session or by means of ICMP echo messages exchanged between our adapter and the server. The total server delay d induced by the page is estimated as the sum of d_i in case HTTP pipelining is not used (and d_i can, therefore, not "overlap" on the server) or by the maximum of d_i, otherwise. Finally, the appropriate `inducedServerLoad` class D is chosen considering the boundary values supplied by the experimenter for the server delay time d.

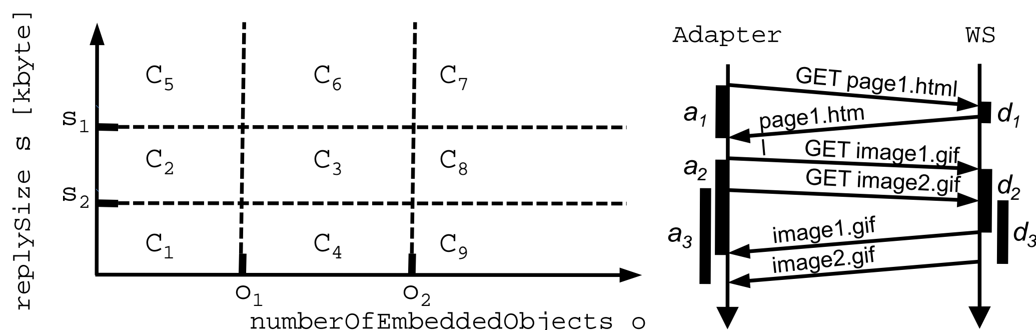

Figure 3 Formation of page `complexity` and `inducedServerLoad` classes.

A series of IE 7.0 browser sessions for a rather small sample of 61 exemplarily chosen home pages were captured and analyzed in [16] using `WireShark` [17] to extract the values

of abstract parameters according to the rules specified above. The IE 7.0 browser was instrumentalized in a manner, that only the object types supported by the adapter were loaded. The results exhibited only marginal difference from the parameter values estimated by the adapter.

6 Construction of the Pool from Popular Web Sites

As already stated in Sec. 5, a sufficiently large, representative and stable pool is indispensable for the generation of realistic and representative Web workloads with `UniLoG`. In this section, we present the construction of a pool populated with home pages extracted from Alexa Web statistics service [18] which ranks Web sites according to their popularity among a large number of users. The representativeness of the Alexa's sample is though restricted to a set of users which were able to install the Alexa's toolbar required to gather the statistics from the users. A total of 1 million most popular sites available for daily download from Alexa's Web site seems to be large enough to construct a comprehensive pool for our load generator, and the restriction to include only the home pages' URLs in the ranking guarantees for some degree of stability in the resulting pool (in respect of reachability of servers and validity of the corresponding pool entries).

The pool was first populated by the top 1000 home pages from the Alexa ranking downloaded from the Alexa Web site on 3 May, 2010, and the values of abstract request parameters were initially set to zero for all pages. The values of real request attributes `serverName`, `serverPort`, `objectName`, `requestMethod`, `optLength` and `optData` were set to the host part of the particular page, `80`, `/`, `GET`, `0` and `NULL`, correspondingly, as the Web sites captured in the Alexa ranking are home pages provided by default on the TCP port 80 by means of a GET request method, and no supplemental data is to be sent to the corresponding Web server by means of POST requests.

Next, `UniLoG.HTTP` was instructed to traverse the pool, retrieve every page and update the values of its abstract request parameters. For this reason, we used the simple UBA presented in Sec. 3, where the values of the `serverName` attribute of every abstract `HttpRequest` to be generated in the R-state $R_{HttpRequest}$ were specified (using the trace parameterization facility of `UniLoG`) to be taken from the column in the pool file, which contains the unique host names of Alexa's home pages. In the adapter, each abstract `HttpRequest` induced a retrieval of the corresponding Web page, whereby the values of its abstract request parameters were estimated and, finally, stored back into the pool.

During the execution of the pool update procedure in May 2010 using the T-Online DSL connection with a maximum of 6 Mbit/s downstream and 640 kbit/s upstream, a total of 979 pool entries were updated. 13 servers were unavailable due to the missing subdomain or Web page to be loaded (as the Alexa ranking contains some pages used only for Webbugs or user tracking). Another 8 sites caused an error during the download by the adapter due to violations of the XHTML or W3C standards in their source code.

A total of 328 servers did not response to ICMP messages making the estimation of the corresponding RTT impossible. The `inducedServerLoad` class for such servers could not be specified ((-1) = n.a., cf. Fig. 4). The mean values of the abstract request attributes `replySize`, `numberObEmbeddedObjects`, `inducedServerLoad`, and `complexity` among all 979 pool entries were estimated to 905.6 kbyte, 38, 2 (delay class "Slow"), and 5 (complexity class C_5), correspondingly. In this exemplarily pool update, we used the threshold values $o_1 = 10$ and $o_2 = 20$ for `numberObEmbeddedObjects` as well as $s_1 = 50$ kbyte and $s_2 = 150$ kbyte for the `replySize` to define the `complexity` class boundaries. For a smaller sample

of Web sites, which were used for testing the implementation of the pool update function presented in Sec. 5, these threshold values provided `complexity` classes with nearly identical number of servers in each class. Applying the same threshold values to Alexa's top 1000 pages emerged that nearly a half of home pages were assigned to the `complexity` class C_7, i.e. these pages contain more than 20 objects and more than 150 kbyte of data (cf. Fig. 4). We note that the experimenter can easily adjust the threshold values in order to obtain `complexity` classes C_1 - C_9 which are suitable for the particular experiment (e.g., classes with uniformly distributed number of servers).

inducedServerLoad d	# servers
-1= n.a. ($d < 0ms$)	328
0 = Immediate ($0ms \leq d < 10ms$)	159
1 = Fast ($10ms \leq d < 30ms$)	66
2 = Slow ($30ms \leq d < 100ms$)	120
3 = Annoyingly slow ($d \geq 100ms$)	306

complexity(numberOfEmbeddedObjects o, replySize s [kbyte])	# servers
$C_1(o \leq 10 \cap s \leq 50)$	149
$C_2(o \leq 10 \cap 50 < s \leq 150)$	73
$C_3(10 < o \leq 20 \cap 50 < s \leq 150)$	65
$C_4(10 < o \leq 20 \cap s \leq 50)$	20
$C_5(o \leq 10 \cap s > 150)$	26
$C_6(10 < o \leq 20 \cap s > 150)$	93
$C_7(o > 20 \cap s > 150)$	504
$C_8(o > 20 \cap 50 < s \leq 150)$	44
$C_9(o > 20 \cap s \leq 50)$	5
n.a. (-1)	0

Figure 4 Number of servers in `inducedServerLoad` and `complexity` classes (cf. [16]).

In this way, the experimenter can create a comprehensive pool of representative Web sites without a lot of effort and in short time. The execution of the pool update procedure immediately before the actual experiment in the particular testbed is a very recommended measure to ensure that the characteristics of Web workloads produced by `UniLoG.HTTP` correspond to the parameter values specified in the pool.

7 Summary and Outlook

In this paper, we presented the application of the `UniLoG` approach to the realistic and rather representative Web workload and traffic generation. Our approach provides different levels of abstraction in the user behaviour models used for load generation. The presented `UniLoG.HTTP` adapter is able to induce Web workloads with characteristics corresponding to the values of abstract parameters of real Web sites from the pool, which is required and provided by our solution. The construction of the sufficiently large, representaive and stable pool of popular Web sites has been presented by us in Sec. 6. The authors hope to have made the next step forward to the combination of Web traffic measurements and generation into a single, coherent approach by this work. The formal load specification technique being used in the `UniLoG` load generator provides for a very precise definition of timing and sequence of resulting requests composing the load being generated, which is indispensable, e.g., to reproduce experiments with a number of predetermined background loads. Moreover, the approach presented in this work is applicable to other interfaces (i.e. TCP, IP, Ethernet) of the protocol stack.

Future work can be directed at the provision of a set of predefined UBA-models for different types and number of HTTP service users. Moreover, the existing adapter prototype can be extended to support a larger set of types for embedded objects or to use a real Web browser (via its remote control interface) for generation and injection of HTTP requests

instead of imitating its behaviour. From the point of view of the authors, these arguments demonstrate the high degree of flexibility and good extensibility of the `UniLoG` approach and support its usage for Web workload generation.

Acknowledgement The authors would like to thank André Gahr and Steffen Jahnke for their valuable contributions to this work in the scope of their diploma theses.

References

1. A. Kolesnikov, M. Kulas, "Load Modeling and Generation for IP-Based Networks: A Unified Approach and Tool Support", Proc. of MMB'2010, Essen, March 15-17, 2010, pp. 91–106.
2. A. Kolesnikov, "Konzeption und Entwicklung eines echtzeitfähigen Lastgenerators für Multimedia-Verkehrsströme in IP-basierten Rechnernetzen", Proc. of Echtzeit'08, Boppard am Rhein, November 27-28, 2008, pp. 91–100.
3. R. Pena-Ortiz et al., "Dweb model: Representing Web 2.0 dynamism", Computer Communications, vol. 32, no. 6, April 2009, pp. 1118–1128.
4. P. Barford, M. Crovella, "Generating Representative Web Workloads for Network and Server Performance Evaluation", Proc. of SIGMETRICS'98, June 24-26, Madison, 1998, pp. 151–160.
5. J. Cao, W. Cleveland, Y. Gao, K. Jeffay, F. Smith, M. Weigle, "Stochastic Models for Generating Synthetic HTTP Source Traffic", Proc. of INFOCOM'04, Hong Kong, March 7-11, 2004, vol. 3, pp. 1546–1557.
6. R. El Abdouni Khayari, M. Rücker, A. Lehmann, A. Musovic, "ParaSynTG: A Parameterized Synthetic Trace Generator for Representation of WWW Traffic", Proc. of SPECTS'08, Edinburgh, June 16-18, 2008, pp. 317–323.
7. K. Vishwanath, A. Vahdat, "Swing: Realistic and Responsive Network Traffic Generation", IEEE/ACM Transactions on Networking, vol. 17, no. 3, June 2009, pp. 712–725.
8. C. Rolland, J. Ridoux, B. Baynat, "Catching IP traffic burstiness with a lightweight generator", Proc. IFIP NETWORKING'07, Atlanta, Mai 14-18, 2007, pp. 924–934.
9. S. Floyd, V. Paxson, "Difficulties in Simulating the Internet", IEEE/ACM Transactions on Networking, vol. 9, no. 4, August 2001, pp. 392–403.
10. M. Arlitt, C. Williamson, "Internet Web Servers: Workload Characterization and Performance Implications", IEEE/ACM Transactions on Networking, vol. 5, no. 5, October 1997, pp. 631–645.
11. A. Williams, M. Arlitt, C. Williamson, K. Barker, "Web workload characterization: Ten years later", Springer, Heidelberg, 2005.
12. S. Avallone et al., "Performance Evaluation of an Open Distributed Platform for Realistic Traffic Generation", Performance Evaluation, vol. 60, 2005, pp. 359–392.
13. J. Wei, C. Xu, "sMonitor: A Non-Intrusive Client-Perceived End-to-End Performance Monitor of Secured Internet Services", Proc. of USENIX'06, Boston, May 30 - June 3, 2006, pp. 243–248.
14. A. Gahr, "Bereitstellung und Einsatz von Modellen und Werkzeugen zur Erzeugung realitätsnaher synthetischer Webserver-Lasten", Diploma Thesis, University of Hamburg, 2008.
15. J. Charzinski, "Traffic Properties, Client Side Cachability and CDN Usage of Popular Web Sites", Proc. of MMB & DFT 2010, Essen, March 15-17, 2010, pp. 136–150.
16. S. Jahnke, "Last-/Verkehrsmessungen und realitätsnahe Lastgenerierung für Web-Server-Zugriffe", Diploma Thesis, University of Hamburg, 2010.
17. WireShark network protocol analyzer, http://www.wireshark.org (16.09.10).
18. Alexa service, http://s3.amazonaws.com/alexa-static/top-1m.csv.zip (16.09.10).