Web Workload Generation Challenges - An Empirical Investigation

Raoufehsadat Hashemian, Diwakar Krishnamurthy, Martin Arlitt

HP Laboratories
HPL-2010-163

Keyword(s):

Workload Generator, Performance Testing, Benchmarking Tools

Abstract:

Workload generators are widely used for testing the performance of Web-based systems. Typically, these tools are also used to collect measurements such as throughput and end user response times that are often used to characterize the Quality of Service (QoS) provided by a system to its users. However, our study finds that Web workload generation is more difficult than it seems. In examining the popular RUBiS client generator [1], we found that reported response times could be grossly inaccurate, and that the generated workloads were less realistic than expected, causing server scalability to be incorrectly estimated. Using experimentation, we demonstrate how the Java Virtual Machine (JVM) and Java network library are the root causes of these issues. Our work serves as an example of how to verify the behaviour of a Web workload generator.

External Posting Date: October 21, 2010 [Fulltext] Approved for External Publication
Internal Posting Date: October 21, 2010 [Fulltext]

Web Workload Generation Challenges - An Empirical Investigation

Raoufehsadat Hashemian[1], Diwakar Krishnamurthy [1], Martin Arlitt[1+2]
[1]. The University of Calgary, Calgary, Alberta, Canada T2N 1N4
[2]. HP Labs, Palo Alto, CA, USA 94304

SUMMARY

Workload generators are widely used for testing the performance of Web-based systems. Typically, these tools are also used to collect measurements such as throughput and end user response times that are often used to characterize the Quality of Service (QoS) provided by a system to its users. However, our study finds that Web workload generation is more difficult than it seems. In examining the popular RUBiS client generator [1], we found that reported response times could be grossly inaccurate, and that the generated workloads were less realistic than expected, causing server scalability to be incorrectly estimated. Using experimentation, we demonstrate how the Java Virtual Machine (JVM) and Java network library are the root causes of these issues. Our work serves as an example of how to verify the behaviour of a Web workload generator.

Keywords: Workload Generator, Performance Testing, Benchmarking Tools

1 INTRODUCTION

Web applications are used by many organizations to provide services to their customers and employees. An important consideration in developing such applications is the Quality of Service (QoS) that the users experience. This motivates the organizations to experimentally evaluate properties such as response time and throughput so that the service can be improved until a desired level of QoS is provided.

Practitioners and researchers rely on benchmarking systems such as RUBiS [1], TPC-W [2] and SPECweb [3] to evaluate the performance of their IT infrastructure before putting it into production. These benchmarking systems typically contain a specially developed Web application. For example, RUBiS provides several different implementations of an online auction application using Web application platforms such as PHP [4] and Enterprise Java Beans (EJB) [5]. These benchmark systems also contain their own workload generator, designed to work specifically with that benchmark application. The *workload generator* is a tool that generates a synthetic workload to the benchmark application to emulate the behaviour of the application's end users. The workload generator reports metrics such as response times for the emulated users and application throughput. These are typically used to assess the QoS provided by the system that executes the benchmark application. In addition to benchmark-specific workload generators, there are numerous general purpose Web request generation tools such as httperf [6], S-Client [7], and JMeter [8]. However, a significant effort may be required to customize these tools to work with a specific application.

A desirable property of a Web workload generator is that it sends and receives requests to a specified Web server in a realistic manner. While this is a simple property to describe, it can be challenging to achieve in practice. Software bottlenecks, hardware bottlenecks and software implementation features pertaining to the workload generation infrastructure can result in unrealistic workloads being generated or inaccurate measurements being recorded.

To minimize the results of measurement errors, a thorough examination of the workload generator is needed. Unfortunately, this aspect is often ignored in practice, thereby putting into risk the validity of the entire performance testing exercise.

In fact, the work described in this document was motivated by concerns about the validity of results from a benchmarking study of a multi-tier Web application system. Specifically, we found the results from an experimental testbed using the RUBiS benchmark were inconsistent across separate invocations of the benchmark. We decided to investigate the causes of such discrepancies through a controlled experimental study.

The details of our study are provided in the remainder of this document. The salient findings of our research are as follows:

- The workload generator can introduce significant errors in the measured end-user response times. The primary cause of this in our experimental environment was the Java Virtual Machine (JVM), which rendered the client host's CPU and memory as the bottleneck, rather than the Web server's resources.
- Multi-threading does not ensure a scalable workload generator. For example, we found the multi-threaded RUBiS workload generator, referred to as *RUBiS client* in this work, supported fewer concurrent users on a client host than the single-threaded httperf tool.
- An unexpected lack of realism in the generated workload can result in the scalability of the Web application being incorrectly estimated. For example, we found that the TCP connection management policy used (via a Java library) by RUBiS client is not appropriate for workload generation. The policy shared a single TCP connection across multiple emulated users. This results in an incorrect estimation of the overhead of TCP connection establishment which a server experiences in practice.

Our work serves as an example of how to verify whether a Web workload generator is behaving as intended. Specifically, we offer a methodology to help recognize problems that could adversely impact the validity of performance testing exercises.

The remainder of this paper is organized as follows. Section 2 provides background information and examines related work. Section 3 describes our experimental environment, including the enhancements made to httperf and RUBiS client to support this work and the network monitoring tools used to validate measurements from the workload generators. Section 4 explains our investigation of the challenges of Web workload generation. Section 5 discusses the implications of our findings. Section 6 summarizes our work and provides directions for future research.

2 BACKGROUND AND RELATED WORK

Workload generators for Web applications rely on the concept of an emulated user. An emulated user submits sequences of inter-dependent requests called *sessions* to a system under study. Dependencies arise because some requests in a session depend on the responses of earlier requests in the session. For example, an order cannot be submitted to an e-commerce system unless the previous requests have resulted in an item being added to the user's shopping cart. This phenomenon is known as an *inter-request dependency*. In each session, an emulated user issues a Web request, waits for the complete response from the system, and then waits for a period of time defined as the *think time* before issuing the next request. A workload generator typically emulates multiple users concurrently. The workload generator runs on one or more *client hosts*, depending on how many emulated users need to be generated.

A challenge for workload generation is ensuring that the synthetic workloads generated are representative of how real users use the system under study. In particular, one must carefully select realistic characteristics for workload attributes that can impact performance, such as the mix of different types of requests submitted to the system, the pattern of arrival of users to the system, and the think times used within user sessions. A discussion on the perils of using incorrect characterizations can be found in [9, 10]. Furthermore, the synthetic workload must preserve the correct inter-request dependencies to stress the system's application functions as intended.

Workload generators can differ in the type of user emulation they support. The most common type of user emulation employed is called the closed approach. With a *closed* approach the number of concurrent users during any given test is kept constant. The next request in a user session is not generated until the previous request has completed and the think time has been observed. The load on the system can be controlled by manipulating the number of concurrent users in a test. With an *open* approach, users submit requests according to a specified rate, without waiting for the response of any of their previous requests that have not completed in the expected time interval. This approach is useful for evaluating Web applications under overload conditions. However, it can violate inter-request dependencies. A hybrid approach combines aspects of the closed and open approaches. With a *hybrid* approach, user sessions are initiated at specified time instants. It is similar to the open approach in that a new session can be initiated before the previous sessions finish. However, similar to the closed approach, within each session a request can only be issued after the response to the previous request in that session has been received. With the hybrid approach the number of concurrent user sessions can change over the course of a test. Schroeder *et al.* argue that the hybrid approach is more representative of real systems than either the closed or open approaches [11].

Workload generators can also differ in the programming paradigms they employ. Most tools follow an approach that uses a combination of multi-threading and synchronous HTTP request-response handling. With this approach each thread independently emulates the behaviour of a user. An alternative approach uses an event-driven mechanism that relies on a single thread and asynchronous HTTP request-response handling. With this approach, a single thread switches between individual users as events such as HTTP requests and responses occur.

Numerous general purpose workload generation tools have been developed. Examples include S-Client [7], httperf [6], SURGE [12], GEIST [13], WAGON [14], JMeter [8], and SWAT [15]. Typically, general purpose workload generators permit workload characteristics to be controlled in a fine grained manner. This allows a system's performance to be studied under many different workloads of interest. While GEIST and S-Client support only the open approach, httperf and SWAT implement the open, closed, and hybrid approaches. The other tools in the list support only the closed approach. S-Client, httperf, and SWAT (which is built on top of httperf) follow the single-threaded, event-based design, while the other tools listed employ the multi-threaded paradigm.

As mentioned in Section 1, practitioners and researchers use Web application benchmark systems such as SPECWeb [3], RUBiS [1] and TPC-W [2] extensively in performance-related studies. The workload generators that are bundled with these benchmark systems are not intended to be as flexible as general purpose tools with regards to fine grained control over workload characteristics. However, they offer an advantage over general purpose tools in that they can be used "off-the-shelf" without the need for time consuming customizations to handle inter-request dependencies of the benchmark applications.

In this study, we focus on the RUBiS client tool, as it was the workload generator we were using when we observed the inconsistent results. For comparison purposes, we enhanced httperf so that it is capable of repeating the workload generated by RUBiS client. RUBiS client [16, 17, 18, 19] and httperf [20, 21, 22, 23, 24] have been used extensively to support experimental performance evaluation of virtualization techniques [17, 21, 23], multi-tier software systems [16, 18, 19, 22] and hardware platforms [20, 24]. The results of tests using these tools are often used in critical decision making. An unrecognized bottleneck caused by the tools or any other unintended behaviour may affect the estimate of performance (i.e., how good is the experience of each user) as well as the estimate of the system's scale (i.e., how many concurrent users can be supported). Consequently, we believe that the outcome of our investigations is likely to be of interest to other researchers and practitioners.

Although benchmark systems are used extensively there is very little work that focuses on systematically evaluating whether these tools work as intended. An exception is Nahum [25], who compared the workload characteristics of the SPECWeb99 benchmark to workloads observed at six production Web server systems. The study found that SPECWeb99 modeled certain characteristics such as file popularity well, but did a poor job of matching characteristics such as file size and HTTP response size. In another study Nahum *et al.* show that not considering workload attributes such as network delays and packet losses during workload generation can result in overly optimistic estimates of server performance [26].

Our work differs from these studies in that it does not focus on validating the choice of workload characteristics or workload attributes used in benchmarks. Instead, we demonstrate and explain why many Java based workload generators can offer misleading performance results. We show that the results from such tools can lead to incorrect conclusions in performance studies.

3 EXPERIMENTAL ENVIRONMENT

In this section we describe our testbed configuration, the network monitoring tools we used to validate the measurements reported by workload generators, and the subtle modifications we made to the workload generators to facilitate a direct comparison between them.

3.1 Testbed Configuration

Our testbed contains three physical machines connected by a 1 Gb/s Ethernet switch. The specifications of these machines are described in Table 1. A schematic of our testbed is shown in Figure 1. One of the three machines is used as the client host while the other two are used as Web and database servers, respectively. The application server is also installed on the Web server machine. The client machine runs a workload generator to exercise the RUBiS auction site installed on the two server machines. Several versions of RUBiS are available using three different technologies namely, PHP, Java Servlets and Enterprise Java Beans (EJB) [1]. Our testbed used the PHP version, which is installed as a Fast CGI module on either the Lighttpd (1.4.26) [27] or the Apache (2.0.63) [28] Web server. Lighttpd is a single process, event-driven, open-source Web server. The Apache Web server provides a group of Multi Processing Modules (MPMs) that allow it to run in a process-based, hybrid (process and thread) or event-hybrid mode. Only one of the Lighttpd or Apache Web servers runs at a time, as required by the specific experiment being run on the testbed. The MySQL database server is installed as the database tier. In our experiments, the Web server machine's CPUs were utilized more than the database server machine's CPUs.

Table 1 Machine Specifications

Properties	Values
Number of Processors	1
Number of Cores	2
Processor Model	Intel Core2 CPU 6400 @ 2.13GHz
Processor Cache Size L1	64 KB
Processor Cache Size L2	2048 KB
Memory Total	2 GB
Memory Swap	2 GB
Kernel	Linux version 2.6.9-55.0.9

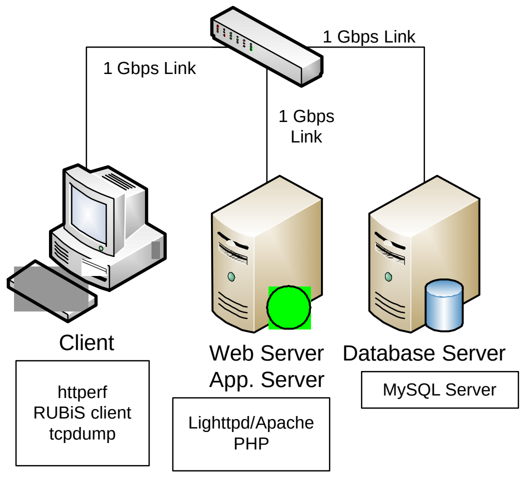

Figure 1 Testbed Configuration

Initially, the RUBiS client workload generator [1] was used to exercise the system. However, as part of our investigation into why the benchmark results were not consistent across runs, we also used the httperf workload generator [6]. Switching to a different workload generator provided insights on causes of the inconsistencies, which we report on in Section 4. To support tests with a large number of concurrent users, we followed the tuning procedure outlined by Brecht [29] that allowed the client machine to maintain a large number of open file descriptors.

3.2 Monitoring Tools

To investigate the causes of the inconsistent benchmark results, we required three sets of data to be monitored. First, we needed the actual client-perceived response times for HTTP requests. In our study we compare these response times to those reported by RUBiS client. Second, we wanted to record the TCP connection establishment and termination events in the system. Last, we needed the resource utilization on the client, Web server and database server.

To verify the HTTP response times as seen by the client, we required an independent monitoring tool which could accurately measure the response times. We achieved this using tcpdump [30], Bro [31] and a Bro script (downloaded from: http://bro-ids.org/bro-contrib/network-analysis/akm-imc05/) that was developed for TCP delay analysis [32]. The script (*reduce.bro*) was modified (renamed as *response.bro*) to calculate the required timings in our experiments. In addition to the actual response time of the Web server, we used these tools to monitor the TCP connection establishment and termination events during each experiment. To obtain these, we modified the "*reduce.bro*" script (renamed as *connection.bro*) so that it tracks the TCP traffic between the client and the server.

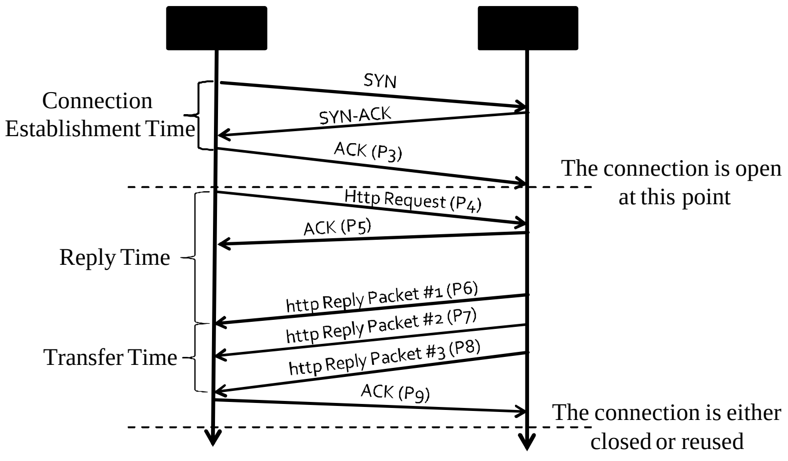
Connection Establishment Time
SYN
SYN-ACK
ACK (P3)
The connection is open at this point
Http Request (P4)
ACK (P5)
Reply Time
http Reply Packet #1 (P6)
http Reply Packet #2 (P7)
http Reply Packet #3 (P8)
Transfer Time
ACK (P9)
The connection is either closed or reused

Figure 2 Typical HTTP Communication

To better understand how we use Bro, consider a typical HTTP request between a client and Web server as shown in Figure 2. The first three packets construct the three-way handshake for establishing a TCP connection. The client sends the connection request to the server in the SYN packet (P1). The servers respond to the connection request by sending the SYN-ACK packet (P2). Then the client sends the ACK packet (P3) as a confirmation to the server and the connection is established. Once the connection is created, the client sends the HTTP request (P4) to the Web server. The server then confirms the arrival of the TCP packet carrying the HTTP request by sending the acknowledgment (P5). After processing the HTTP request, the HTTP reply is sent in one or more TCP packets, depending on the size of requested object. In Figure 2, the server sends the reply in three packets (P6, P7 and P8). Once the client receives all of the reply packets, it acknowledges receipt of the last packet and the HTTP request is complete (P9). If the TCP connection is persistent, it can be used for other HTTP request. Additional HTTP requests on this connection avoid the establishment overhead (i.e., the connection establishment time) as well as associated overheads (e.g., TCP slow-start). The HTTP 1.1 protocol allows persistent TCP connections to be exploited [33].

The time to complete the entire HTTP request can be broken into two parts, as shown in Figure 2. The first part is the difference between the time the first reply packet is received at the client and the time at which the client issued the HTTP request. We refer to this time as the *reply time*. The second interval is the duration of time for the entire reply to be transferred to the client. We call this interval *transfer time*. We use the sum of the reply time and transfer time as *response time* for evaluating the accuracy and representativeness of the workload generators.

Before each experiment, tcpdump is started and configured to capture all HTTP packets (TCP port 80). The tcpdump command line options we used to capture the network traffic are as follows:

- **tcpdump** ip host *xxx.xxx.xxx.xxx* and tcp port *80* -w *outputfile*
  - The "ip host" directive specifies that only IP packets "from" or "to" the specified *xxx.xxx.xxx.xxx* host address should be captured. For our tests "*xxx.xxx.xxx.xxx*" can refer to the IP address of either the Web server or the client machine.
  - The "-w" option writes the raw packets to *outputfile* rather than parsing and printing them to the console.

When each experiment ends, tcpdump is stopped and its output file is given as the input to Bro. The primary purpose of Bro is as an intrusion detection system (IDS), but its powerful analysis capabilities make it attractive for our purposes. The modified Bro script (*response.bro*) calculates the reply times and transfer times of HTTP requests and records them in its own output file. While Bro can run directly on live network traffic, we run it in an off-line manner to reduce the overhead on the client.

We describe the Bro command line options to run the *response.bro* script as follows:

- **bro** -r *outputfile* response.bro > *response.csv* 2> *response.err*
  - The "–r" option specifies the name of the input tcpdump (pcap) format trace file to read and analyze.
  - "*response.bro*" is the name of Bro script that specifies the analysis to conduct.
  - "*response.csv*" is an output file that contains the HTTP response times calculated by Bro for each HTTP request submitted during the test.
  - "*response.err*" is an output file that captures any warnings generated by Bro.

The fields recorded by the output file (*response.csv*) of the modified Bro script are as follows.

- **response.csv** : "*End of transaction time stamp*, *Port*, *Connection ID*, *Request #*, *Pipelined flag*, *Reply time*, *Response time*"
  - *End of transaction time stamp*: The time stamp which is recorded when the HTTP request is completed.
  - *Port*: The TCP port used by client to communicate with the server.
  - *Connection ID*: An identifier for the TCP connection used to send the request.
  - *Request #:* Number of HTTP requests which have been sent through this TCP connection before current request.

- *Pipelined*[1] *flag:* Specifies whether the request pipelining used in this connection (1) or not (0).
- *Reply time*[2]*:* The difference between the time at which the client issued the HTTP request and the time when first reply packet is received at the client.
- *Response time:* The sum of reply time and transfer time.

The *connection.bro* script can be executed similar to how the *response.bro* script is invoked. This script outputs a file called *connection.csv* that includes the following information:

- **connection.csv**: "*Event time stamp, Event flag, Number of open connections*"
  - *Event time stamp*: The time stamp which is recorded at the completion of an event (TCP connection establishment and connection close).
  - *Event flag*: If set to '1' the event was "Connection Establishment" and if set to "0" it was a "Connection termination" event. The total number of connections established during the experiment can be calculated by counting number of '1' values in this column.
  - *Number of open connections*: The number of established connections which are open at this instant.

There are several alternatives for executing tcpdump. For example, tcpdump can be run on a dedicated machine. That machine could receive a mirrored copy of the traffic from the Ethernet switch (e.g., from a switch that supports port mirroring). The main advantage of this approach is that there is no overhead on the client machine. A disadvantage is that it would have slightly less accurate measurements of when the client received each packet. For our study, we ran tcpdump on the client machine during our experiments. This enables us to get the best estimate of the response times as experienced by the client. However, it does place some load on the resources of the client machine. To ensure that the overhead of tcpdump does not affect the test results, a set of simple tests were conducted and repeated with and without tcpdump. We found that for our tests, tcpdump did not affect the accuracy of measured response times. However, tcpdump could affect measured response times in some situations (e.g., high network utilization), so care must be taken when using it.

Since the server and client machines used the Linux operating system, the *sysstat* [34] package was used to monitor the resource utilizations on each machine. The *sysstat* package contains the *sar*, *sadf*, *iostat*, *pidstat* and *mpstat* commands for Linux. The *sar* command collects and reports system activity information. The information collected by *sar* can be saved in a file in binary format for future inspection. We used the CPU, memory, network interface and swap space metrics supported by the package to monitor how busy each machine was. The sampling interval was set to 1 second for all the performance counters used by this study.

[1] According to the HTTP protocol specification: "A client that supports persistent connections MAY "pipeline" its requests (i.e., send multiple requests without waiting for each response). A server MUST send its responses to those requests in the same order that the requests were received." [32]

[2] The response times in "response.bro" are measured with an accuracy of 10 microseconds.

3.3 Workload Generator Modifications

As mentioned earlier, RUBiS client was the main target of this study. We also used httperf to independently validate results reported by RUBiS client. Although RUBiS client and httperf both emulate users' HTTP transactions with a Web application, they have different capabilities and support different workload specifications. Therefore, we needed to make some minor changes in both applications so that their results would be directly comparable. In the following sections the modifications made to each workload generator are discussed.

3.3.1 RUBiS client Modifications

RUBiS client is a Java-based tool which emulates user sessions using Java threads. The test specifications are defined in the "*rubis.properties*" file. We made the following four changes to RUBiS client:

1. The original RUBiS client application supports only closed session-based workloads. Since we wanted to have the option of submitting a more realistic workload to our Web server, we added the capability of creating sessions in a hybrid manner to RUBiS client. The inter arrival times between successive new sessions are specified in a text file and the file path is added to the *rubis.properties* file. Alternatively, our modifications also allow a tester to specify exponentially distributed session inter arrival times with a specified mean.
2. For our experiments, we required a record of each HTTP request and its response time (measured by RUBiS client), as well as a mapping of the request to the session that generated it. Since the original RUBiS client did not provide this feature, we customized the code to record this information in a log file. This file also records when each HTTP request gets submitted by RUBiS.
3. The original RUBiS client measures the response time in milliseconds. However, for some of the requests in our testbed, the response time was less than one millisecond. Consequently, the RUBiS output statistics were inaccurate in low load conditions, i.e., many response times were reported as 0 ms. To calculate response time more accurately, we used *System.nanoTime()* [35] instead of the *System.currentTimeMillis()* [36] Java function. As the name implies, *System.nanoTime()* returns timestamps with nanosecond precision.
4. To simplify the experiments and knowing the fact that in most real systems a separate server is used as the "image server", we disabled the image downloading part of the RUBiS code.

It is important to note that none of these modifications affect the general performance of the RUBiS client, while change #3 improves the accuracy of its response time measurements.

3.3.2 httperf Modifications

httperf is an event-based, single-threaded tool which is capable of emulating different types of workloads. With httperf, test parameters are specified as command line options. An example httperf invocation is as follows:

- **httperf** --server=www.example.com --port=80 --wsesslog *2000*,0,*session_file* --period=*d*, 0.001

In this command, httperf submits requests to "*www.example.com*" at TCP port "80". The "*wsesslog*" option indicates that the workload to be generated is session-based. *2000* sessions are to be generated as per the session definitions found in the *"session_file"* text file. Each session definition in this file is a list of the URIs to be requested from the Web server with successive requests in the session definition separated by a think time. The "*period*" option specifies properties of the inter arrival time between successive new sessions generated by httperf. The first parameter specifies the inter arrival time distribution. The other parameter in the "*period*" option specifies the mean value for the session inter arrival time distributions [37]. In the example shown previously, the inter arrival times are set to a deterministic, i.e., constant, value by selecting the "*d*" option and this constant value is set to 1 millisecond. Closed workloads can be realized with httperf by specifying a very low mean session inter arrival time and a large number of requests within each session.

For each experiment in this study, we conduct a test with RUBiS client followed by a test with httperf. To facilitate comparison, we require httperf to submit the same workload to the system under test as was submitted by RUBiS client. In particular, the sequence of sessions submitted by RUBiS client and httperf needs to be identical. This is achieved by extracting the session sequence and think time sequence within each session from the log file generated from the RUBiS client test (modification 2 of Section 3.3.1) and saving that information to a "*session_file*" for use with httperf. Furthermore, the sequence of inter arrival times between successive new sessions generated by RUBiS client needs to be preserved during the httperf test. This requirement caused us to modify httperf to enhance the *"period"* option. The modification allows a user to specify a sequence of inter arrival time values as input to httperf. In the following example, httperf uses a new "*s*" switch with the *"period"* option to submit sessions as per the sequence of inter arrival times specified in the "*inter_arrival_file*":

- --period=*s,inter_arrival_file*

In our experiments, we first extract the sequence of session inter arrival times from a RUBiS test by parsing the log file generated by RUBiS client. We save this sequence in an *"inter_arrival_file"* and specify that file as input with the new *"period"* option. These modifications ensure that the sequence of sessions and the instants at which new sessions are generated (relative to the start of the test) are identical in the RUBiS client and httperf tests in an experiment. Furthermore, a session generated by httperf at a given time instant observes the same sequence of think times as the session generated by RUBiS client at the same time instant.

We have also added a new option to httperf for logging detailed information for individual HTTP requests submitted in a test. Specifically, the new "--rfile_name=*file_name"* option can be used to save the detailed information to the text file specified by "*file_name*". The information recorded includes the time instant at which a request was submitted as well as the reply time and transfer time recorded by httperf for that request. The values are recorded with 1 microsecond precision.

4 INVESTIGATION WEB WORKLOAD GENERATION CHALLENGES

As mentioned in Section 1, the goal of this study is to understand some inconsistencies observed in an earlier RUBiS benchmarking study. We started our exploration by attempting to validate the accuracy and correctness of the experiment results reported by RUBiS client. To achieve this, we needed to measure the client-observed response times independent of the

RUBiS client. We did this by monitoring the network traffic and extracting the timing information of HTTP transactions, using the tools and methods described in Section 3.2. In addition to verifying the accuracy of the response time measurements, we also wanted to verify whether RUBiS client emulates users in the expected manner. As mentioned in the previous section, we achieved this by conducting a separate set of tests using a second workload generator (httperf). The measurements collected using this methodology allowed us to identify the root causes of the problems we are investigating. The following subsections present the detailed results of this exploration process.

Section 4.1 characterizes the accuracy of RUBiS client when it uses the default GNU JVM bundled with many Linux installations. Section 4.2 repeats the analysis with the latest version of the JVM released by Sun that exploits the "HotSpot" [38] technology. In Section 4.3 we explore factors that limit the scalability of workload generators. Finally, Section 4.4 highlights how unrealistic TCP connection management policies can cause a workload generator to provide misleading insights into the scalability of the system under study.

4.1 Validating Accuracy of Measured End-User Response Times

To validate the accuracy of response times reported by RUBiS client we conducted a set of experiments using the Lighttpd Web server. Since the original RUBiS client is only capable of generating closed workloads, we use a closed workload initially. The workload causes a mix of browse, buy, and sell transactions to be submitted to the RUBiS application. We set the think time distribution to a negative exponential [39] with a mean value of 7 seconds, consistent with the RUBiS specifications. The variable factor for these experiments is the number of concurrent user sessions in the system. We vary this parameter to achieve different utilization levels for the bottleneck resource in the server machines as well as the client machine. As described previously, in each experiment we first conduct a test with RUBiS client. We then use httperf to submit the same workload generated by RUBiS client using the approach described in Section 3.3.2.We record the response times reported by each of the two workload generators and the actual Web server response times measured by Bro. We define the difference between the mean response time reported by a workload generator and the actual mean response time reported by Bro as the *absolute error* of the workload generator.

We performed seven tests with each of the workload generators, varying the number of concurrent user sessions *N* from 500 to 4,000. Beyond 4,000 user sessions, the RUBiS client reported "out of memory" exceptions. This indicates that the JVM could not obtain enough memory from the operating system to spawn the requisite number of Java threads for user emulation.

Figures 3(a) and 3(b) show the absolute error of RUBiS client and httperf, respectively as a function of the Web server's CPU utilization. It must be noted that Figure 3(a) and Figure 3(b) have very different scales for the Y-axis. Furthermore, both workload generators cause approximately the same utilization on the Web server's CPUs, for a given value of *N*. This behaviour is along expected since both tools submit the same workload to the server.

Since Bro's measurements are based on packet traces, we expect its response times to be slightly lower than what RUBiS client would see higher up the TCP/IP stack. However, the results in Figure 3(a) show that the increases are much larger than we would have anticipated. Even for light workloads (*N*=500, mean Web server CPU utilization of 2.5%) the RUBiS client reports the mean response time 500 microseconds higher than what Bro reports. With this absolute error, the RUBiS client mean response time estimate is 1.07 times the actual Bro measured mean response time. As *N* increases, the discrepancy increases significantly. For example at *N*=4,000 the Web server CPU utilization is 22% and the difference between the mean response times reported by RUBiS and Bro is around 150 milliseconds. For this case, the RUBiS client mean response time is 2.6 times the Bro response time. In contrast, the absolute error values for httperf do not change much with *N* and vary between 10 to 40 microseconds. Most of this discrepancy is likely due to Bro measuring in the lower layers of network protocol stack. The absolute error in the first experiment (*N*=500) is more than 15 times higher in RUBiS client compared to httperf. The gap between RUBiS client and httperf in terms of error increases considerably as the workload intensity increases and the Web server is more utilized. The httperf results reveal that the RUBiS client errors are not due to a common bottleneck like the network or the server. We note that the maximum mean Web server CPU utilization observed during the tests is only 22.2% at *N*=4,000.

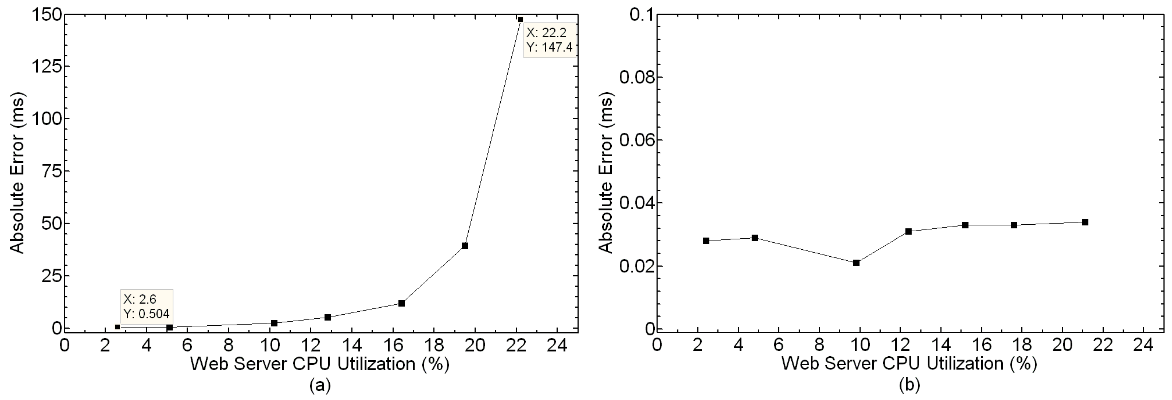

Figure 3 Absolute Error (a) RUBiS client–GNU JVM (b) httperf

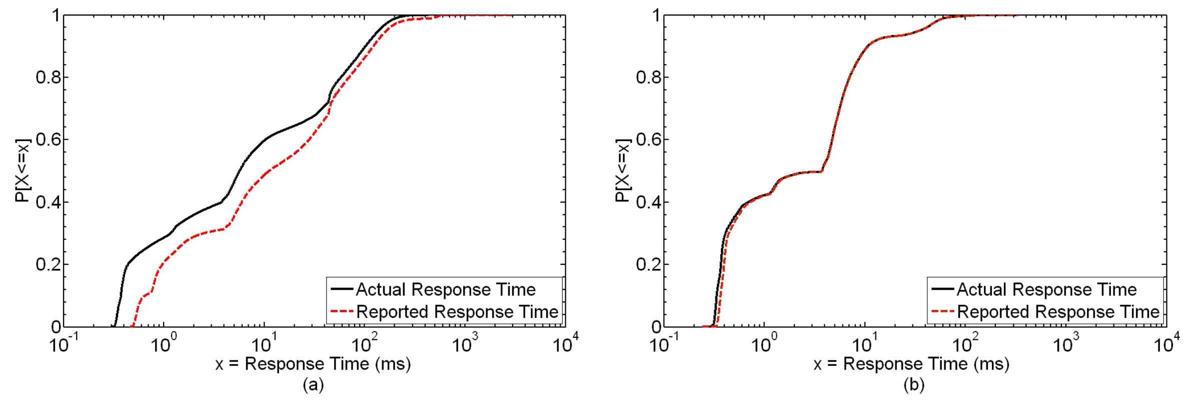

Figure 4 CDF of Response Times (*N*=3000) - (a) RUBiS client-GNU JVM (b) httperf

To better understand how often the discrepancies occur, we plot the cumulative distribution function (CDF) of the response times. Figures 4(a) and 4(b) show the CDF plots for both httperf and RUBiS client tests for the *N*=3,000 case. Figure 4(a) shows that there are significant discrepancies between the Bro measured response times and RUBiS client reported response times in both the high and low response time ranges. In contrast, Figure 4(b) shows that the two CDFs are almost indistinguishable for httperf. We therefore conclude that RUBiS client significantly overestimates server response times while such a problem is not evident with httperf.

We now investigate possible causes for the inaccurate RUBiS client response time measurements. Figure 5 shows the client machine's mean CPU utilization when running the RUBiS client tests for different numbers of concurrent user sessions. The plot shows that the client machine's CPU utilization increases rapidly with an increase in the number of concurrent user sessions. The figure indicates that the client CPU utilization is strongly related to the number of threads created to emulate users. Figures 3 and 5 also jointly indicate that the CPU of the client machine is the bottleneck in the experimental setup. For example, the mean utilization of the Web server CPU with *N*=4,000 is 22.2% (Figure 3(a)) whereas the mean utilization of the client machine's CPU at this setting is 60% (Figure 5). Therefore, it is likely that this bottleneck in the client side is the main source of the large absolute error values in the RUBiS client results.

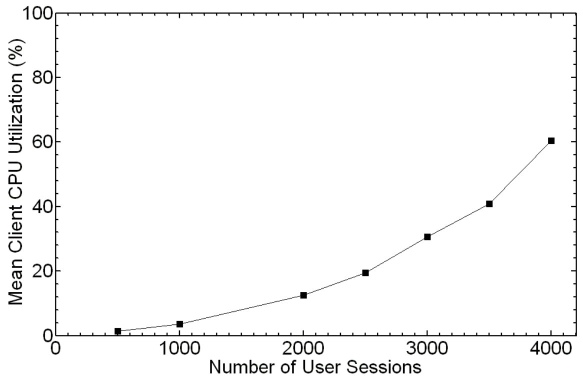

Figure 5 Mean client machine CPU utilization for RUBiS client-GNU JVM

4.2 Changing the JVM

On closer inspection, we realized that the RUBiS client experiments described in Section 4.1 used the default GNU JVM (GNU libgcj version 4.1.2 released in 2007) [40] which is bundled with some Linux distributions. We decided to experiment with a more recent JVM and hence selected the latest version of Sun's JVM for Linux (version 1.6-18 released February 2010). A key difference between the two JVMs is that the Sun JVM includes the "HotSpot" [38] technology designed to improve performance.

In experiments using the Sun HotSpot JVM, we varied the number of user sessions from 500 to 5,100. Beyond 5,100 users, the RUBiS client encountered "out of memory" exceptions. We investigate this issue further in Section 4.3. As with the previous section, we used the Lighttpd Web server for these experiments.

Figure 6 shows the mean client machine CPU utilization for these experiments as well as the experiments with the GNU JVM. The figure reveals that the Sun HotSpot JVM utilizes the client machine's CPUs considerably less than the GNU JVM. For instance, while the CPU in the client machine is on average 30% utilized for *N*=3,000 in the GNU JVM, this value is decreased to 5 % with Sun HotSpot JVM.

The accuracy of RUBiS client also improves when using Sun HotSpot JVM. Figures 7(a) and 7(b) show the values of absolute error for different mean Web server CPU utilization levels for RUBiS client on Sun HotSpot and httperf, respectively. For RUBiS client using the Sun HotSpot JVM, the absolute error is less than 3 milliseconds up to *N=5,000*. However, the absolute errors are still noticeably higher compared to the corresponding httperf absolute errors. Moreover, the absolute error values are almost constant before *N*=5,000, which indicates that intensifying the workload by increasing the number of users does not have a corresponding effect on the magnitude of absolute error. There is a steep increase in absolute error for *N*=5,000 and *N*=5,100. From Figure 7, the absolute errors for *N*=5,000 and *N*=5,100 are 70 ms and 230 ms, respectively. As mentioned, we were unable to go farther than 5,100 users because of the out-of-memory exception encountered by RUBiS client. The largest and smallest factors by which RUBiS client mean response times exceeded Bro measured mean response times in these experiments were 8.149 and 1.061, respectively. In contrast, the worst case for httperf was one where its response time estimate was 1.008 times the corresponding Bro response time.

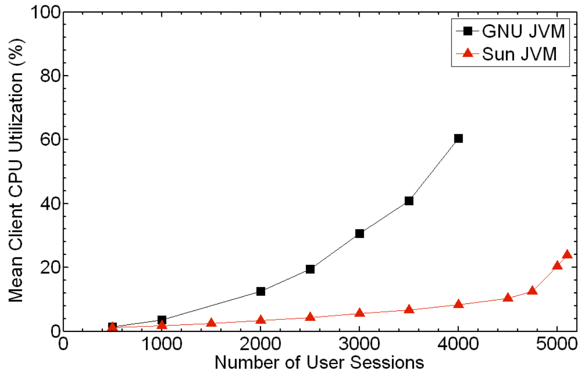

Figure 6 Mean client machine CPU utilization for RUBiS client tests with both JVMs

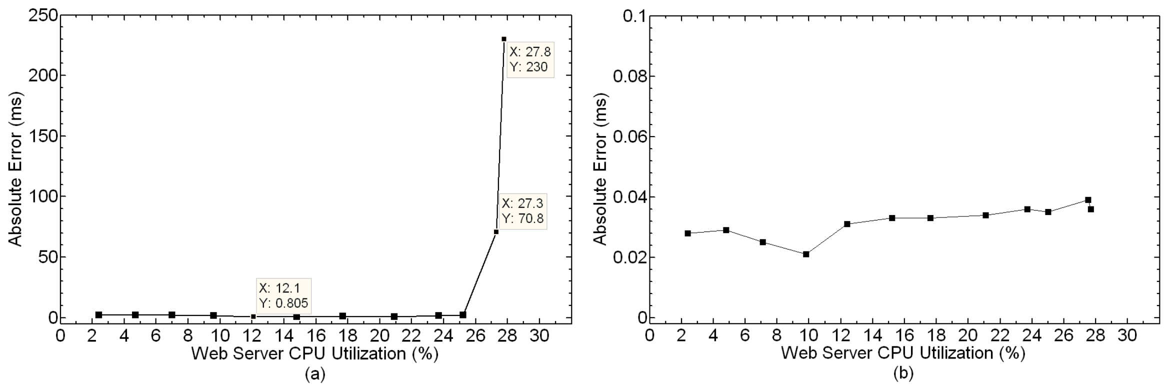

Figure 7 Absolute Error (a) RUBiS client–Sun JVM (b) httperf

Figure 8 shows the CDF plot for one of the RUBiS client tests using Sun HotSpot JVM (*N*=3,000). The mismatch between the CDF of the actual response times and the CDF of response times reported by RUBiS client is decreased compared to the RUBiS client experiments using GNU JVM (Figure 4(a)). With the Sun HotSpot JVM, the discrepancy only appears for lower response times. For response times that are less than 1 ms, there is a considerable difference between the Bro measured response times and RUBiS client reported response times.

These results reveal that the performance of the JVM affects the accuracy of results reported by RUBiS client. However, it is not clear whether this aspect gets attention during performance testing exercises. Care must be exercised to ensure that a high performance JVM is used during benchmarking studies. When a test system has multiple JVMs installed, for example for backward compatibility with other applications on the system, one must ensure that the appropriate JVM is invoked when executing tools such as RUBiS client. For example, it might be safer to specify the full pathname of the high performance JVM executable to avoid the possibility of invoking an inappropriate JVM.

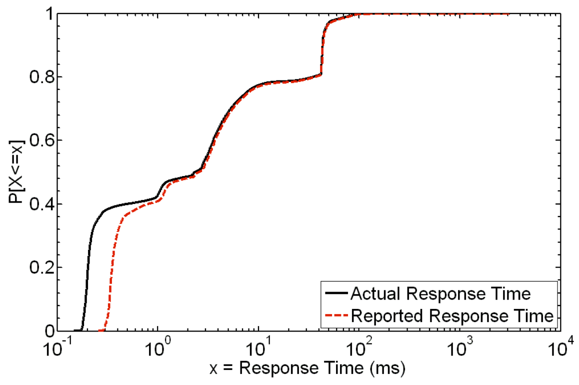

Figure 8 CDF of Response Times for RUBiS client-Sun JVM (*N*=3,000)

4.3 Investigating the Scalability of Workload Generators

In this stage of our investigation, we searched for the bottleneck in the client machine which causes RUBiS client a large absolute error in mean response time for *N*=5,000 and *N*=5,100 and out-of-memory exceptions for values greater than 5,100. Looking at the resource utilization information gathered from the client machine, we found out that the low (default) value of the maximum heap size used by the JVM creates this bottleneck. We used the Java command line option "–Xmx=*heapsize*" to increase the maximum heap size for the JVM. While the default heap size used by Sun HotSpot JVM[3] was 64 MB we set this value to 512MB and repeated some of the experiments with this new value. It should be mentioned that with 2GB of total physical memory on the client machine, 512 MB was the largest maximum heap size we could specify while following the recommendations of the Java performance tuning best practices document [41]. The Lighttpd server was used for the tests presented in this section.

The new heap size increased the capacity of RUBiS client from 5,100 to 6,300 threads. The absolute error in RUBiS client reported mean response time dropped to 1.049 ms for *N*=5,000 and 1.093 ms for *N*=5,100 in the new configuration. Beyond 6,300 threads, we encountered the out-of-memory exception again which prevented us from using RUBiS client to further increase the load on the server.

We now compare the scalability of RUBiS client and httperf based on the results presented so far. Figure 9 plots the actual, i.e., Bro measured, mean response times for various values

[3] This setting pertains to the JVM optimized for "client" code.

of *N* for RUBiS client and httperf. The maximum number of users that RUBiS client could emulate on the client node was 6,300 with the Sun HotSpot JVM and 512 MB heap size setting. At this setting the mean utilization of the Web server CPUs was 36%. The memory bottleneck at the client machine prevents RUBiS client from stressing the server further. In contrast, we were able to conduct tests with up to *N*=8,000 concurrent users with httperf as shown in Figure 9. At this setting, httperf was able to drive the utilization of the Web server CPU to up to 90%. Furthermore, the mean response time reported by httperf was very accurate for this case. For *N* values greater than 8,000 the server was severely loaded leading to unstable behaviour such as very long request response times and a large number of connection resets and timeouts.

These results indicate that multi-threading does not automatically ensure a scalable workload generator. We note that httperf was configured to exploit only one of the two available processor cores of the client machine in our experiments. In contrast, RUBiS client used both cores. This suggests that the single-threaded, event-driven httperf tool can support significantly more number of emulated users on a single host than RUBiS client.

4.4 Effect of TCP Connection Management Policy on the Validity of Performance Results

For the remainder of the paper we consider only the Web server response times measured by Bro, to facilitate a direct comparison. From Figure 9 we observe that the RUBiS client with GNU JVM, RUBiS client with Sun HotSpot JVM, and httperf yield very different mean server response times even though they have been configured to submit the same workload. For example, with *N*=3,000 the server response times are 31.6, 8.3 and 4.2 milliseconds respectively with RUBiS client (GNU JVM), RUBiS client (Sun HotSpot JVM), and httperf. The rest of this section explains the differences and their implications to server performance evaluation.

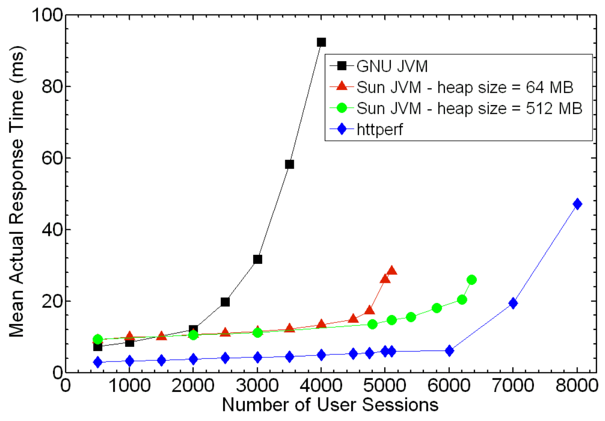

Figure 9 Mean Actual Response Time of Web server for all workload generators

After a detailed examination of the behaviours of RUBiS client and httperf, we realized that the primary difference between them was due to differences in the TCP connection management policy used in each application. On paper, both claimed to open a single (dedicated) persistent TCP connection per user session (see RUBiS documentation [42] and httperf man page). However, our test results revealed significant differences in the way httperf and RUBiS client handle persistent connections. While we found that httperf exhibits

its documented behaviour, we observed that RUBiS client handles the TCP connections quite differently due to its use of certain functions in the Java network library. Specifically, a thread first creates a *URL* object, which then calls the *URL.openStream()* [43] Java function. This function returns an *InputStream* object whose methods are called by the thread to read the HTTP response pertaining to the *URL*. After reading the response, the thread closes the *InputStream* object by calling its *close()* method. Java documentation states that this method releases any system resources associated with the stream [44]. The ensuing results show that the use of these functions causes RUBiS client to deviate from its intended behaviour of using one dedicated connection per user session.

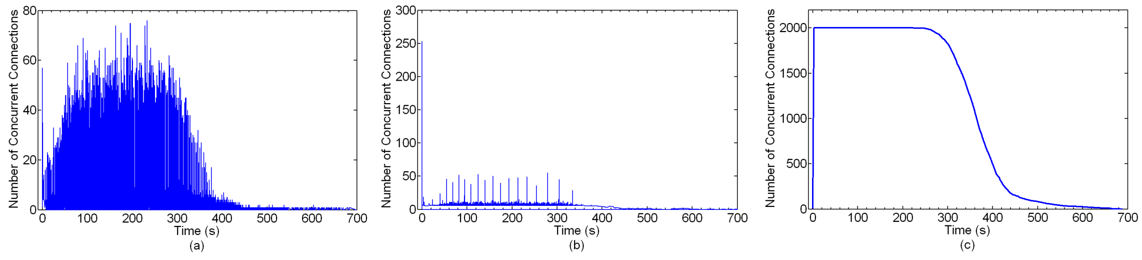

Figure 10 Number of Concurrent Connections - (a) RUBiS client-GNU JVM (b) RUBiS client-Sun JVM (c) httperf

To illustrate the differences between the TCP connection management policies, we present in Figure 10 a time series of the number of concurrent connections observed during a test with both httperf and RUBiS client for one of the experiments presented in the earlier sections (*N*=2,000). This data was obtained by using the *connection.bro* script described in Section 3.2. In the RUBiS client tests (Figures 10(a) and 10(b)), the total number of concurrent connections fluctuates from 0 to 75 for the GNU JVM and 0 to 250 for the Sun HotSpot JVM. These maximum values are much less than the number of concurrent user sessions (*N*=2,000). In contrast, the httperf test (Figure 10(c)) sees the number of concurrent TCP connections jump from 0 to 2,000 at the beginning of the experiment[4]. After around 300 seconds, the number of concurrent connections starts to decrease as sessions begin to complete.

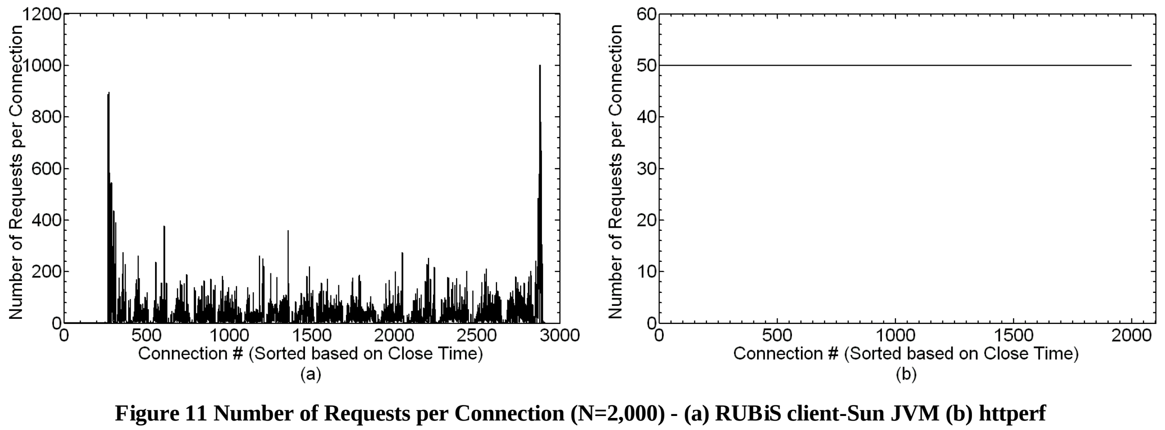

Figure 11 Number of Requests per Connection (N=2,000) - (a) RUBiS client-Sun JVM (b) httperf

[4] Since it is a closed workload, all the sessions start at the beginning of the experiment.

Figure 11 provides further evidence of the differences in the way persistent connections are handled by RUBiS client and httperf. Figures 11(a) and 11(b) show the number of requests sent through each unique TCP connection used during RUBiS client and httperf tests, respectively. The x axis of these figures show connections sorted based on the time they were closed. Figures 11(a) and 11(b) show that both workload generators used persistent TCP connections since each connection was used to issue multiple requests. With httperf, 2,000 connections were opened during the test and each connection submitted 50 requests as shown in Figure 11(b). The number of requests submitted per connection corresponds to the number of requests per session which was configured to be 50 for both workloads. In contrast, with RUBiS client the number of connections used during a test is 2,912 which is greater than the number of concurrent sessions. *This indicates that there are more connection establishment and connection shutdown activities in the RUBiS client workload.* From Figure 11(a), at the beginning of the RUBiS experiment a large number of connections (around 250) are opened to send the first requests for the 2,000 users. These connections are closed after 1 or 2 HTTP requests were submitted through them. However, the number of requests submitted per connection for a vast majority of connections is significantly greater than 50, the number of requests per session. The number of requests submitted in a connection was as high as 1,000. Figure 10 and Figure 11 together establish that RUBiS client *causes multiple user sessions to use a single connection.* In effect a small number of concurrent TCP connections are shared across a large number of emulated users.

The higher server response times observed with RUBiS client in Figure 9 are likely due to the higher connection establishment and connection shutdown overheads in RUBiS client. Specifically, the highest response time in Figure 9 was caused by GNU JVM which established the most number of connections. The Sun HotSpot JVM established a lesser number of connections and caused requests to incur lower response times at the server. httperf established the least number of connections and caused the least stress on the server.

The sharing of TCP connections across users in RUBiS client is likely not to be representative of the behaviour observed in real systems because clients do not typically initiate requests from the same connection. We conducted additional experiments to better understand how the connection management policies used by the tools impact server behaviour. Specifically, we constructed hybrid workloads to emulate a “flash crowd” scenario, i.e., a sudden increase in the rate of arrival of sessions, using httperf and RUBiS client. We then studied the behaviour of the Lighttpd and Apache servers under these workloads. We configured Apache to use the “prefork” module [45] for request processing. Recalling from Section 3, Lighttpd is an event-based server that uses an asynchronous mechanism to handle HTTP requests. It is lightweight in that it is designed to use just a single process per processor. For all our experiments we configured Lighttpd to use two processes [46]. The prefork module of Apache maintains a pool of worker processes with each process handling an incoming connection in a synchronous manner. When the number of incoming connections exceeds the number of worker processes, Apache spawns additional processes to handle the increased load. For our experiments we used the default *Apache* setting where the initial size of the worker process pool is 15.

Figure 12(a) shows the non-bursty and the bursty, i.e., flash crowd workloads used for Lighttpd. From Figure 12(a), in the non-bursty workload the sessions arrive with a mean rate of 10 sessions per second. In the bursty workload, the session arrival rate increases suddenly from 10 to 333 causing a burst of sessions over a 4 second time interval. Table 2 shows the mean response times measured by Bro for both workloads while using httperf and RUBiS client. Under both the workload generators, Lighttpd is able to handle the bursty workload with only a marginal increase in mean response time. However, the mean response times with httperf are lower than the corresponding response times with RUBiS client. This is consistent with the behaviour observed in the previous experiments which also used Lighttpd. The extra connection establishment and connection disconnection overheads imposed by RUBiS client seems to dominate this scenario.

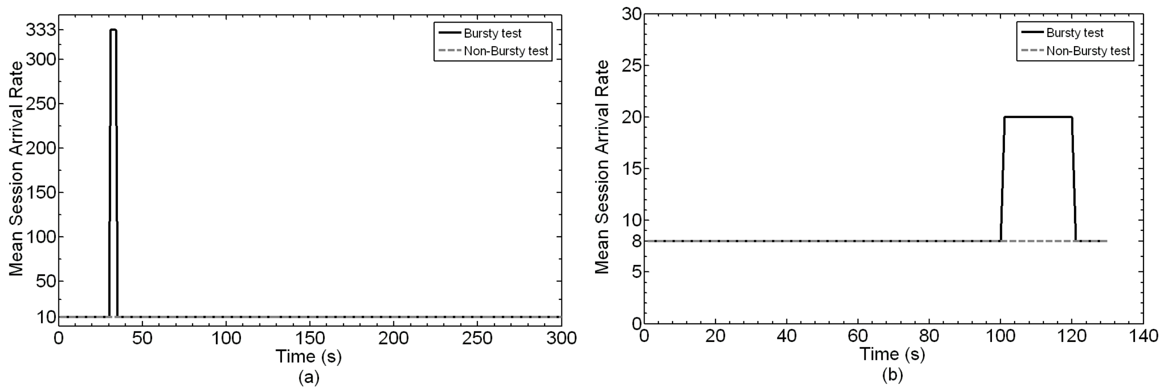

Figure 12 Session Arrival Rate (Sessions/Second) -(a) Lighttpd (b) Apache

Figure 12(b) shows the non-bursty and bursty workloads used for Apache. Since the Apache and Lighttpd servers have dissimilar overheads for handling requests from new TCP connections, we had to apply a workload with a different burst specification. Figure 12(b) shows that in the non-bursty workload sessions arrive at the rate of 8 sessions per second. In the bursty workload, the session arrival rate suddenly increases from 8 to 20 causing a burst of sessions to arrive over a 20 second time interval.

Table 2 Bursty test results

		Apache Response Time (ms)			**Lighttpd Response Time (ms)**		
		Median	**Mean**	**95 percentile**	**Median**	**Mean**	**95 percentile**
RUBiS client	***Non-Bursty***	2.118	4.333	23.334	2.243	11.003	44.995
	Bursty	2.135	4.571	23.755	2.411	12.369	48.384
httperf	***Non-Bursty***	2.164	11.537	27.878	2.150	4.061	20.588
	Bursty	2.233	58.647	33.457	2.165	4.643	20.488

From Table 2, data obtained from RUBiS client tests on Apache shows that there is very little increase in the mean, median and 95 percentile of response times from the non-bursty to bursty case. This may lead a tester to reach the conclusion that the Apache Web server is scalable with respect to handling bursts in session arrivals. However, the use of httperf provides a diametrically opposing viewpoint regarding the server's scalability. While using httperf, the mean response time for the bursty workload is more than 5 times the mean response time of the non-bursty workload. The 95th percentile of response time is also considerably higher for the bursty workload while the median response time is not affected much. These observations imply that a small fraction of requests in the bursty workload encountered very high response times. The results show that burstiness has a significant detrimental impact on Apache's performance. Interestingly, while RUBiS client places lesser stress than httperf on Apache the situation is reversed for Lighttpd. This suggests that a factor other than connection establishment and connection shutdown overheads dominates server performance in this scenario.

The reason for the long response times encountered while using httperf to generate a bursty workload to Apache can be explained as follows. To generate the burst of sessions shown in Figure 12(b), httperf initiates a large number of concurrent connections, one per each new session in the burst, to the server. This causes the total number of concurrent connections issued by httperf to Apache to increase beyond the worker process pool size. As a result a number of sessions in the burst encounter significant delays related to the time Apache takes to spawn new worker processes to handle the increase in load. From Table 2, this type of performance degradation was also observed with the non-bursty workload although to a smaller extent. RUBiS client does not cause the bottleneck in worker process pool size to be exposed due to its sharing of connections across multiple users. The maximum number of concurrent connections issued by RUBiS client during the test was always less than the Apache worker process pool size. Table 2 reveals that Lighttpd is less sensitive to burstiness under both httperf and RUBiS client. Due to its event-driven, asynchronous request processing architecture, it avoids overheads related to spawning new processes to handle a burst of incoming connections.

Summarizing, the choice of unrealistic TCP connection management policies can provide misleading insights into server performance and scalability. For example, the policy of the Java library used by RUBiS client provided pessimistic estimates of performance for Lighttpd but overly optimistic estimates for Apache with respect to the more realistic one dedicated connection per session policy used by httperf. We note that while httperf uses a more realistic policy than that of the Java library used by RUBiS client, many modern browsers use more than one connection per session. For example, Souders reports that Web browsers like Internet Explorer 8 and Firefox 3 use up to six parallel TCP connections to transfer HTTP transactions [47]. Workload generators must consider such complexities to ensure the validity of performance testing exercises.

5 DISCUSSION

Benchmarking computer systems is a challenging task, as there are many possible mistakes that can be made [39]. One common mistake listed by Jain is “not validating measurements” [39]. The solution to this is to cross-check the measurements, which is an approach we used in this work. By using this approach, we revealed that the RUBiS client was incorrectly reporting the performance and scalability of the Web server under test, due to limitations of the JVM and the Java networking library used by the RUBiS client workload generator. Our specific implementation of the approach is straightforward to apply in other Web server benchmarking studies. It is important to note that the overhead of the approach should be quantified (as we did) in each case, to avoid another common benchmarking mistake [39].

Code reuse is a common practice in software development, as it can dramatically reduce the time (and therefore cost) to develop an application. However, a disadvantage of reusing source code is that any problems that exist with the initial code can propagate to other applications. This issue is relevant to our work, as the TPC-W workload generator shares a similar implementation to the RUBiS client generator. In particular, both are implemented in

Java, support a closed workload approach, follow the multi-threaded paradigm, and employ the same TCP connection management policy (via a common Java library) described in Section 4.4. As a result, we expect studies that have used either RUBiS or TPC-W to benchmark a Web server may have incorrectly estimated the performance or scalability of the server, for the reasons discussed in Section 4.

6 CONCLUSION

This paper described our experience in validating the performance and scalability results reported by the RUBiS client Web workload generator. After observing inconsistent benchmarking results with RUBiS client, we implemented a method to cross-check the results. This uncovered two root causes for the inconsistent results: the JVM and the Java network library used by the generator. We also showed that a multi-threaded workload generator is not necessarily more scalable than an efficiently implemented, event-based, single-threaded generator.

Due to the importance of Web workload generation, we believe that similar validation work should be conducted for other common workload generators. In particular, we plan a similar study for the SPECWeb workload generator, as it is commonly used in industry to benchmark cutting edge servers. Since the results of such studies are used for purposes such as purchase decisions, the ramifications for inaccurate benchmark results are potentially more significant.

Source code for our workload generator enhancements, the modified Bro monitoring scripts, and further details on our experimentation methodology can be found at: http://people.ucalgary.ca/~dkrishna/SPE

ACKNOWLEDGMENTS

This work was financially supported by Natural Sciences and Engineering Research Council (NSERC) Canada and Hewlett-Packard (HP). The authors would like to thank Jerry Rolia of HP Labs for his helpful suggestions and feedback.

REFERENCES

1. RUBiS – Homepage. http://rubis.ow2.org/ [01 October 2010]
2. TPC-W- Homepage. http://www.tpc.org/tpcw/ [01 October 2010]
3. SPEC- Benchmarks. http://www.spec.org/benchmarks.html#web [01 October 2010]
4. PHP: Hypertext Processor. http://www.php.net/ [01 October 2010]
5. Enterprise Java Bean Technology. http://www.oracle.com/technetwork/java/index-jsp-140203.html [01 October 2010]
6. Mosberger D, Jin T. httperf: A tool for measuring web server performance. *ACM SIGMETRICS Performance Evaluation Review* 1998; 26(3): 31-37. DOI: http://doi.acm.org/10.1145/306225.306235
7. Banga G, Druschel P. Measuring the capacity of a web server under realistic loads. *World Wide Web* 1999; 2(1): 69–83. DOI: 10.1023/A:1019292504731
8. Apache JMeter. http://jakarta.apache.org/jmeter/ [01 October 2010]
9. Feitelson D. The Forgotten Factor: Facts on Performance Evaluation and its Dependence on Workloads. *Proceeding of Int. Euro-Par Conference* 2002; 2400: 49-60.
10. Paxson V, Floyd S. Wide Area Traffic: The Failure of Poisson Modeling. *IEEE/ACM Transactions on Networking* 1995; 3(3): 226-244. DOI: http://doi.acm.org/10.1145/190314.190338

11. Schroeder B, Wierman A, Harchol-Balter M. Open versus closed: A cautionary tale. *Proceedings of the 3rd Conference on Networked Systems Design & Implementation* 2006; 3: 18-18.
12. Barford P, Crovella M. The surge traffic generator: Generating representative web workloads for network and server performance evaluation. *Proceedings of the ACM SIGMETRICS* 1998; 151-160. DOI: http://doi.acm.org/10.1145/277851.277897
13. Kant K, Tewari V, Iyer R. GEIST: Generator of ecommerce and internet server traffic. *Proceedings of Int. Symposium on Performance Analysis of Systems and Software* 2001; 49-56.
14. Liu Z, Niclausse N, Jalpa–Villanueva C. Traffic model and performance evaluation of web servers. *Performance Evaluation* 2001; 46(2-3):77–100. DOI: http://dx.doi.org/10.1016/S0166-5316(01)00046-3
15. Krishnamurthy D, Rolia J, Majumdar S. A Synthetic Workload Generation Technique for Stress Testing Session-Based Systems, *Proceedings of the IEEE Transactions on Software Engineering* 2006; 32(11), 868-882. DOI: http://doi.ieeecomputersociety.org/10.1109/TSE.2006.106
16. Guitart J, Carrera D, Torres J, Ayguadé E, Labarta J. Tuning Dynamic Web Applications using Fine-Grain Analysis. *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing* 2005; 84-91. DOI: 10.1109/EMPDP.2005.44
17. Padala P, Zhu X, Wang Z, Singhal S, Shin K. Performance Evaluation of Virtualization Technologies for Server Consolidation. *Technical Report HPL-2007-59* 2007; DOI: 10.1.1.70.4605
18. Malkowski S, Hedwig M, Pu C. Experimental evaluation of N-tier systems: Observation and analysis of multi-bottlenecks. *Proceedings of the IEEE international Symposium on Workload Characterization* 2009; 118-127. DOI: http://dx.doi.org/10.1109/IISWC.2009.5306791
19. Sicard S, Boyer F, De Palma N. Using components for architecture-based management: the self-repair case. *Proceedings of the 30th international Conference on Software Engineering* 2008; 101-110. DOI: http://doi.acm.org/10.1145/1368088.1368103
20. Guitart J, Carrera D, Beltran V, Torres, J, Ayguadé E. Dynamic CPU provisioning for self-managed secure web applications in SMP hosting platforms. *Computer Networks* 2008; 52(7), 1390-1409. DOI: http://dx.doi.org/10.1016/j.comnet.2007.12.009
21. Wood T, Cherkasova L, Ozonat K, Shenoy P. Profiling and modeling resource usage of virtualized applications. *Proceedings of the 9th ACM/IFIP/USENIX international Conference on Middleware* 2008; 366-387. DOI: 10.1007/978-3-540-89856-6_19
22. Rolia J, Krishnamurthy D, Casale G, Dawson S. BAP: a benchmark-driven algebraic method for the performance engineering of customized services. *Proceedings of the First Joint WOSP/SIPEW international Conference on Performance Engineering* 2010; 3-14. DOI: http://doi.acm.org/10.1145/1712605.1712609
23. Kusic D, Kephart J.O, Kandasamy N, Jiang G. Power and Performance Management of Virtualized Computing Environments Via Lookahead Control. *Cluster Computing* 2009; 12(1), 1-15. DOI: http://dx.doi.org/10.1007/s10586-008-0070-y
24. Ramamurthy P, Sekar V, Akella A, Krishnamurthy B, Shaikh A. Remote profiling of resource constraints of web servers using mini-flash crowds. *USENIX Annual Technical Conference on Annual Technical Conference* 2008; 185-198. DOI: 10.1.1.145.4897
25. Nahum E. Deconstructing SPECweb99. *7th International Workshop on Web Content Caching and Distribution (WCW)* 2002.
26. Nahum E, Rosu M, Seshan S, Almeida J. The effects of wide-area conditions on WWW server performance. *Proceedings of the ACM SIGMETRICS international Conference on Measurement and Modeling of Computer Systems* 2001; 257-267. DOI: http://doi.acm.org/10.1145/378420.378790
27. Lighttpd fly light. http://www.lighttpd.net/ [01 October 2010]
28. The Apache HTTP Server Project. http://httpd.apache.org/ [01 October 2010]
29. Brecht T. “Linux: Increasing the number of open file descriptors”. Online tutorial available at: http://www.cs.uwaterloo.ca/~brecht/servers/openfiles.html [01 October 2010]
30. TCPDUMP/LIBPCAP public repository. http://www.tcpdump.org/ [01 October 2010]
31. Paxson V. Bro: A system for detecting network intruders in real-time. *Computer Networks* 1999; 31(23-24): 2435–2463. DOI: 10.1016/S1389-1286(99)00112-7
32. Arlitt M, Krishnamurthy B, Mogul J.C. Predicting short-transfer latency from TCP arcana: A trace-based validation. *Proceedings of the 5th ACM SIGCOMM Conference on internet Measurement* 2005; 19-19.
33. Hypertext Transfer Protocol -- HTTP/1.1. http://www.w3.org/Protocols/rfc2616/rfc2616.html [01 October 2010]
34. SYSSTAT. http://pagesperso-orange.fr/sebastien.godard/ [01 October 2010]
35. Java Documentation Link. http://download.oracle.com/Javase/1.5.0/docs/api/Java/lang/System.html#nanoTime%28%29 [01 October 2010]
36. Java Documentation Link. http://download.oracle.com/javase/1.4.2/docs/api/java/lang/System.html#currentTimeMillis%28%29 [01 October 2010]
37. httperf Manual. http://www.hpl.hp.com/research/linux/httperf/httperf-man-0.9.pdf [01 October 2010]
38. Oracle (Sun) Java HotSpot Technology, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136373.html [01 October 2010]

39. Jain R. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons: New York, 1991
40. GCJ: The GNU Compiler for Java-GNU Project - Free Software Foundation (FSF). http://gcc.gnu.org/java/ [01 October 2010]
41. Java Performance Tuning. *Sun White Paper* 2005; Available at: http://java.sun.com/performance/reference/whitepapers/tuning.html [01 October 2010]
42. Amza C, Ch A, Cox A, Elnikety S, Gil R, Rajamani K, Cecchet E, Marguerite J. Specification and implementation of dynamic Web site benchmarks. *Proceedings of WWC-5: IEEE 5th Annual Workshop on Workload Characterization* 2002; 3-13.
43. Java Documentation Link. http://Java.sun.com/j2se/1.4.2/docs/api/Java/net/URL.html#openStream%28%29 [01 October 2010]
44. Java Documentation Link. http://Java.sun.com/j2se/1.4.2/docs/api/Java/io/InputStream.html#close%28%29 [01 October 2010]
45. Prefork - Apache HTTP Server. http://httpd.apache.org/docs/2.0/mod/prefork.html [01 October 2010]
46. Lighttpd - Server.max-workerDetails - lighty labs. http://redmine.lighttpd.net/projects/lighttpd/wiki/Server.max-workerDetails [01 October 2010]
47. Souders S. Roundup on Parallel Connections. Available at: http://www.stevesouders.com/blog/2008/03/20/roundup-on-parallel-connections/ [01 October 2010]