



## Red Hat – Industry Standard Benchmarks

# SPECweb<sup>®</sup> 2009 Benchmark using Red Hat<sup>®</sup> Enterprise Linux<sup>®</sup> 5.3 on a HP ProLiant<sup>®</sup> DL370 G6 (Result = 100,051 sus @ 410 W)

Accoria Network's  
Rock Web Server v1.4.7,  
Rock JSP/Servlet Container v1.3.2

Red Hat<sup>®</sup> Enterprise Linux<sup>®</sup> 5.3

HP ProLiant DL370 G6  
(2 x 4 = 8 Core Intel Xeon W5580 Nehalem)

10-Gigabit NICs

Version 1.0  
August 2009





**SPECweb2009<sup>®</sup> Benchmark using  
Red Hat<sup>®</sup> Enterprise Linux<sup>®</sup> 5.3  
on a HP Proliant<sup>®</sup> DL570 G6  
(Result = 100,051 sus @410 W)**

1801 Varsity Drive  
Raleigh NC 27606-2072 USA  
Phone: +1 919 754 3700  
Phone: 888 733 4281  
Fax: +1 919 754 3701  
PO Box 13588  
Research Triangle Park NC 27709 USA

Linux is a registered trademark of Linus Torvalds. Red Hat, Red Hat Enterprise Linux and the Red Hat "Shadowman" logo are registered trademarks of Red Hat, Inc. in the United States and other countries.

SPECweb@99 is a registered trademark of the Standard Performance Evaluation Corporation (SPEC).

All other trademarks referenced herein are the property of their respective owners.

© 2009 by Red Hat, Inc. This material may be distributed only subject to the terms and conditions set forth in the Open Publication License, V1.0 or later (the latest version is presently available at <http://www.opencontent.org/openpub/>).

The information contained herein is subject to change without notice. Red Hat, Inc. shall not be liable for technical or editorial errors or omissions contained herein.

Distribution of modified versions of this document is prohibited without the explicit permission of Red Hat Inc.

Distribution of this work or derivative of this work in any standard (paper) book form for commercial purposes is prohibited unless prior permission is obtained from Red Hat Inc.

The GPG fingerprint of the [security@redhat.com](mailto:security@redhat.com) key is:  
CA 20 86 86 2B D6 9D FC 65 F6 EC C4 21 91 80 CD DB 42 A6 0E



# Table of Contents

1. Executive Summary .....	5
2. Overview of SPECweb2009.....	6
2.1 Logical Components of SPECweb2009 .....	6
2.1.1 Client .....	7
2.1.2 Prime Client .....	7
2.1.3 Web Server.....	7
2.1.4 Back-End Simulator (BeSim) .....	8
2.1.5 Power Analyzers.....	8
2.1.6 PTDaemon .....	8
2.1.7 Temperature Sensor.....	8
2.1.8 Storage .....	9
2.2 Performance Metrics .....	9
2.2.1 Component Workloads .....	9
2.2.2 Primary Metrics (SPECweb2009_(JSP/PHP)_Peak and SPECweb2009_(JSP/PHP)_Power).....	10
2.2.3 Workload Submetrics.....	11
2.3 SPECweb2009 Internals .....	11
2.3.1 Architecture .....	11
2.3.2 Implementation .....	12
2.3.2.1 Application Workload Timing.....	13
2.3.2.2 Power Workload Timing .....	15
2.3.3 Harness Software Design .....	16
SPECweb2009 Client Harness Flow Overview:.....	17
LoadGenerator .....	17
WorkloadScheduler.....	18
Connection .....	19
BankingWorkloadScheduler's makeRequest() method .....	19
SPECweb2009 Client Harness Flow Diagram: .....	21
SPECweb 2009 Prime Client Process Overview:.....	22
SPECweb.....	22



SPECwebControl .....	22
specwebPTDInterface .....	23
RemoteLoadGenerator .....	23
2.3.4 Rated Receive and Quality of Service .....	26
2.3.5 Think Time and HTTP 304 Server Responses .....	26
2.3.6 Workload Design .....	27
2.3.7 Harness Configuration Files .....	27
2.4 Conclusion .....	28
2.5 Additional Information.....	28
3. System Configuration.....	29
4. SPECweb2005 Performance Results .....	32
5. References .....	34
Appendix: Positioning Industry Standard Benchmarks .....	35



# 1. Executive Summary

SPEC has defined the SPECweb2009 benchmark, the successor to the SPECweb2005 benchmark. Red Hat Enterprise Linux has maintained a consistent lead in SPECweb2005 benchmark results. Now, all six of the published SPECweb2009 results reported so far have been on Red Hat Enterprise Linux. This SPECweb2009 results described below uses Red Hat Enterprise Linux (RHEL) 5.3 running on a HP ProLiant DL370 G6 (based on 2 socket x 4 cores/socket = 8 core Intel Xeon W5580 Nehalem processor) to achieve the **best SPECweb2009 result to date**.

<b>Hewlett-Packard: HP ProLiant DL370 G6</b>	<b>SPECweb2009_JSP_Peak = 100,051 sus @ 410 W</b>	
	SPECweb2009_JSP_Banking = 108,096 sus @ 419 W	
	SPECweb2009_JSP_Ecommerce = 139,200 sus @ 430 W	
	SPECweb2009_JSP_Support = 66,560 sus @ 379 W	
	<b>SPECweb2009_JSP_Power = 212 asus/watt</b>	
<b>Red Hat Enterprise Linux 5.3</b>		
<b>Accoria Networks, Inc.: Rock Web Server v1.4.7 (x86_64)</b>		
<b>Accoria Networks, Inc.: Rock JSP/Servlet Container v1.3.2 (x86_64)</b>		
<b>Tested By:</b> Hewlett-Packard	<b>SPEC License #:</b> 3	<b>Test Date:</b> Jul-2009

This benchmark result is a demonstration of the close and continued cooperation between Hewlett Packard Company, Intel and Red Hat Inc. to showcase the superior combined performance of Red Hat Enterprise Linux (RHEL) running on HP's Intel Xeon-based ProLiant servers. For the latest SPECweb2005 benchmark results, visit:

<http://www.spec.org/web2009/results/web2009jsp.html>

or

<http://www.spec.org/web2009/results/web2009php.html>



## 2. Overview of SPECweb2009

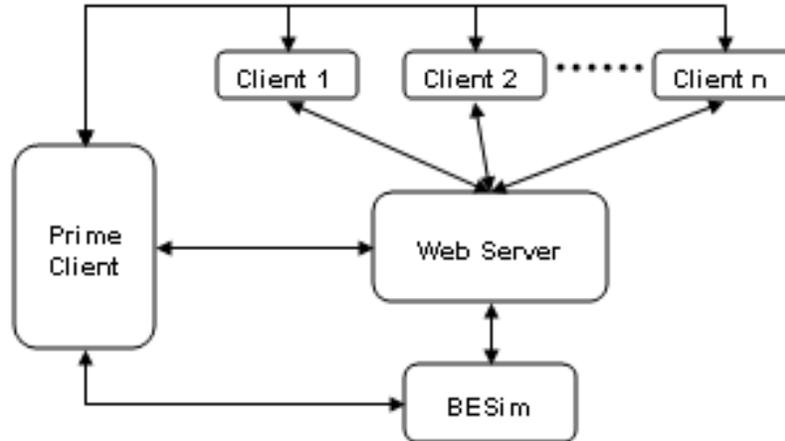
SPECweb2009 is a software benchmark product developed by the Standard Performance Evaluation Corporation (SPEC), a non-profit group of computer vendors, system integrators, universities, research organizations, publishers, and consultants. It is **designed to measure a system's ability to act as a web server servicing static and dynamic page requests.**

SPECweb2009 is the successor of SPECweb2005. Rather than offering a single benchmark workload that attempts to approximate the breadth of web server workload characteristics found today, SPECweb2009 has chosen a 4-workload benchmark design: banking, ecommerce, support, and power. The change to the workloads in SPECweb2009 over SPECweb2005 is the addition of the Power workload and the inclusion of power measurement methodology ( [http://www.spec.org/power\\_ssj2008/docs/SPECpower-Methodology.pdf](http://www.spec.org/power_ssj2008/docs/SPECpower-Methodology.pdf) ). The benchmark is capable of measuring both SSL and non-SSL request/response performance, and continues the tradition of giving Web users the most objective and most representative benchmark for measuring web server performance as well as power. SPECweb2009 disclosures are governed by an extensive set of run rules to ensure fairness of results.

This section will discuss the benchmark architecture and performance metrics. Separate design documents offer workload-specific design details.

### 2.1 Logical Components of SPECweb2009

SPECweb2009 has four major logical components: the clients, the prime client, the web server, and the back-end simulator (BeSim). These logical components of the benchmark are illustrated, below:



**Logical Components of SPECweb2009**

## 2.1.1 Client

The benchmark clients run the application program that sends HTTP requests to the server and receives HTTP responses from the server. For portability, this application program and the prime client program have been written in Java. Note that as a logical component, one or more load-generating clients may exist on a single physical system.

## 2.1.2 Prime Client

The prime client initializes and controls the behavior of the clients, runs initialization routines against the web server and BeSim, and collects and stores the results of the benchmark tests. Similarly, as a logical component it may be located on a separate physical system from the clients, or it may be on the same physical system as one of the clients.

## 2.1.3 Web Server

The web server is that collection of hardware and software that handles the requests issued by the clients. In this documentation, we shall refer to it as the SUT (System Under Test) or Web Server. The HTTP server software may also be referred to as the HTTP daemon.



## 2.1.4 Back-End Simulator (BeSim)

BeSim is intended to emulate a back-end application server that the web server must communicate with in order to retrieve specific information needed to complete an HTTP response (customer data, for example). BeSim exists in order to emulate this type of communication between a web server and a back-end server. BeSim design documentation is located here:

<http://www.spec.org/web2009/docs/design/BeSimDesign.html>

## 2.1.5 Power Analyzers

Power Analyzer is used to measure and collect data on Voltage, Current and Watts used by the SUT. A typical setup will use two Power Analyzers, one connected to the System and the other to the storage subsystem. It is however possible to have a set up with no separate storage subsystem (where all the data is stored internally), in which case there would be only one power analyzer used. The Power Analyzers feed the information to the PTDaemon.

## 2.1.6 PTDaemon

The power/temperature daemon (also known as PTDaemon, PTD or ptd) is used by SPECweb2009 to offload the work of controlling a power analyzer or temperature sensor during measurement intervals to a system other than the SUT. It hides the details of different power analyzer interface protocols and behaviors from the benchmark software, presenting a common TCP-IP-based interface.

The SPECweb2009 harness connects to PTD by opening a TCP port. For configurations including separate storage, multiple IP/port combinations can be used to control multiple devices.

PTD can connect to a variety of analyzer and sensor types, via protocols and interfaces specific to each device type. The device type is specified by a parameter passed locally on the command line on initial invocation of the daemon.

The communication protocol between the SUT and PTD does not change regardless of device type.

## 2.1.7 Temperature Sensor

The Temperature Sensors measure the temperature at the air inlet to the SUT. Multiple sensors may be required when using separate storage, in particular when the storage device does not share the same rack as the system tested.. The information collected by the Temperature sensor is passed to the PTDaemon.



## 2.1.8 Storage

Storage subsystems may be separate from the server. When using separate storage, the power information for the Storage has to be monitored using a different power analyzer and collected by PTDaemon.

## 2.2 Performance Metrics

The SPECweb2009 benchmark has two primary metrics; SPECweb2009\_(JSP/PHP)\_Peak and SPECweb2009\_(JSP/PHP)\_Power. Additionally, there are submetric scores for the three workloads, Banking, Ecommerce, and Support. The primary performance metric SPECweb2009\_(JSP/PHP)\_Peak is expressed in the form X simultaneous user sessions @ Y Watts. Here, X is the geometric mean of the **simultaneous user sessions (sus)** for the three workloads and Y represents the geomean of the corresponding Watts recorded. The other primary metric SPECweb2009\_(JSP/PHP)\_Power is expressed in the units of **sus/Watt** and is derived as the ratio of the sum of simultaneous sessions to the sum of the average watts measured at different load levels while running the Power workload. It is important to understand the significance of both the primary metric scores and the submetric scores for SPECweb2009 results.

### 2.2.1 Component Workloads

The SPECweb2005 benchmark consists of three workloads: Banking, Ecommerce, and Support, each with different workload characteristics representing common use cases for web servers. Each workload measures the number of simultaneous user sessions a web server can support while still meeting stringent quality-of-service and error-rate requirements. The aggregate metric reported by the SPECweb2005 benchmark is a normalized metric based on the performance scores obtained on all three workloads.

Component	Description
Banking	Models online banking. Represents number of customers accessing accounts at a given time that



	can be supported with acceptable Quality of Service.
E-Commerce	Models an online retail store.  Of the three workloads, Ecommerce workload probably fits the profile of most customers. This is because unlike Banking, and Support workloads, this workload is a mixture of HTTP and HTTPS requests. The I/O characteristics fall in the between Banking and Support workloads.
Support	Represents users acquiring patches and downloads from a support web site.  The support workload is the most I/O intensive of all the workloads.

## 2.2.2 Primary Metrics (SPECweb2009\_(JSP/PHP)\_Peak and SPECweb2009\_(JSP/PHP)\_Power)

The formulae used to compute the metrics SPECweb2009\_(JSP/PHP)\_Peak and SPECweb2009\_(JSP/PHP)\_Power are as shown below.

SPECweb2009\_Peak is thus derived from the three submetrics, as a geometric mean. The Watts corresponding to this is also derived as a geometric mean of the Watts used while running the workloads for Banking, Ecommerce and Support. We note that the Watts here represents the power used by the system as well as storage

While SPECweb2009\_(JSP/PHP)\_Power is defined as the ratio of the sum of simultaneous user sessions to the sum of average watts over various steps of the Power workload, for a valid run, the maximum simultaneous sessions used for Power should be the same as the sus for Ecommerce. Since a valid run uses 100%, 80%, 60%, 40%,20% and 0% of the maximum load, the numerator in this case is essentially the same as SPECweb2009\_(JSP/PHP)\_Ecommerce  $\times (1 + 0.8 + 0.6 + 0.4 + 0.2 + 0) = \text{SPECweb2009\_Ecommerce} \times 3$ .



$$\text{SPECweb2009\_ (JSP/PHP)\_Peak} = (\text{SPECweb2009\_Banking} * \text{SPECweb2009\_Ecommerce} * \text{SPECweb2009\_Support})^{(1/3)} \text{ sus}$$

$$\text{@ Watts} = (\text{Watts\_Banking} * \text{Watts\_Ecommerce} * \text{Watts\_Support})^{(1/3)}$$

$$\text{SPECweb2009\_ (JSP/PHP)\_Power} = \frac{(1+0.8+0.6+0.4+0.2+0) * \text{SPECweb2009\_Ecommerce}}{\Sigma \text{ Watts}} \text{ asus/watt}$$

## 2.2.3 Workload Submetrics

Workload submetric scores are in units of simultaneous sessions. They represent the number of simultaneous user sessions the SUT was able to support while meeting the quality-of-service (QOS) requirements of the benchmark. While the two primary metrics offer performance and power efficiency scores for the system being tested, the submetric scores offer a workload-by-workload view of a system's performance characteristics and power usage at peak levels.

Detailed power and temperature data are collected and reported in results files.

## 2.3 SPECweb2009 Internals

### 2.3.1 Architecture

The SPECweb2009 benchmark is used to measure the performance of HTTP servers. The HTTP server workload is driven by one or more client systems, and controlled by the prime client. Each client sends HTTP requests to the server and validates the server responses. When all of the HTTP requests have been sent and responses received that constitute a web page (typically, this is a dynamic response plus any embedded image files), the number of bytes received, the response time, and the QOS criteria met for that web page transaction is recorded by the client.

Prior to the start of the benchmark, one or more client processes, is started on each of the client systems. The PTDaemon is started and initialized for all measurement devices. Client processes either listen on the default port (1099) or on another port specified by the user in Test.config. Once all client processes have been started, the client systems are ready for workload and run-specific initialization by the prime client.



The prime client will read in the key value pairs from the configuration files, Test.config, Testbed.config, and the workload-specific configuration file (ex: SPECweb\_Banking.config), and perform initialization for the web server and for BeSim. Upon successful completion, it will initialize each client process, passing each client process the configuration information read from the configuration files, as well as any configuration information the prime client calculated (number of load generating threads, for example). When all initialization has completed successfully, the prime client will start the benchmark run.

At the end of the benchmark run, the prime client collects this result data from all clients and power and temperature data from the PTDaemon instances, aggregates this data, and writes this information to a results file. When all three iterations (or a single iteration in the case of Power workload) have finished, an ASCII text report file and an HTML report file are also generated.

## 2.3.2 Implementation

The number of simultaneous sessions corresponds to the number of load-generating processes/threads that will continuously send requests to the HTTP server during the benchmark run. Each of these threads will start a "user session" that will traverse a series of workload-dependent states. Once the user session ends, the thread will start a new user session and repeat this process. This process is intended to represent users entering a site, making a series of HTTP requests of the server, and then leaving the site. A new user session starts as soon as the previous user session ends, and this process continues until the benchmark run is complete.

A load-generating thread will make a dynamic request to the HTTP server on one connection, and then will reuse that connection as well as an additional connection to the server to make parallel image requests for that page. This is intended to emulate the common browser behavior of using multiple connections to request the page image files from the server. Note that the load-generating thread does not extract the images from the web page returned, as that would create unnecessary page parsing burden for the load-generating threads. Instead, the page image files to be requested for each dynamic page are retrieved from the workload-specific configuration file.

For each page requested by a load-generating thread, the load generator will start a timer immediately before sending the page request to the HTTP server, and it will stop the timer as soon as the last byte of the response for that page is received. It will likewise time the responses for all supporting image files and add those response times

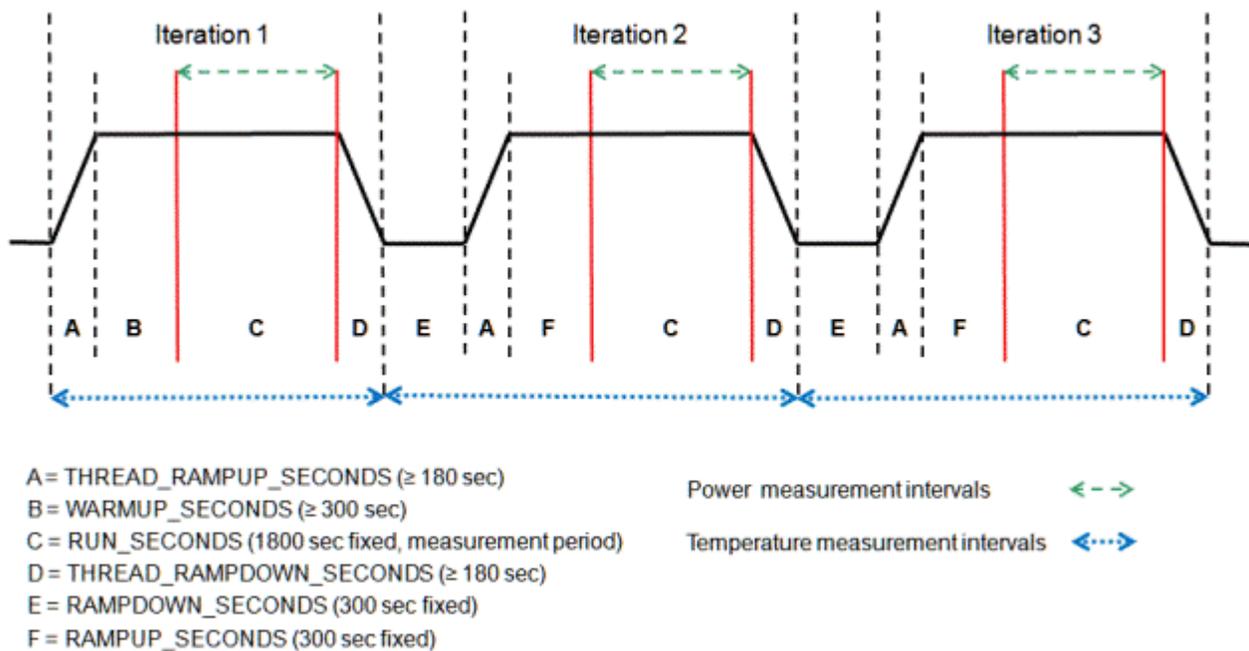


to the dynamic page response time in order to arrive at a total response time for that page that includes all supporting image files. Valid responses will then have their aggregate page response time checked against their respective QOS values for the workload, and the value for the corresponding QOS field (TIME\_GOOD, TIME\_TOLERABLE, or TIME\_FAIL) will be incremented.

At the end of a run, the prime client aggregates the run data from all of the clients and determines whether the run met the benchmark QOS criteria.

The prime client controls the phases of the benchmark run. These phases are illustrated in the diagram, below:

### 2.3.2.1 Application Workload Timing



### SPECweb2009 Benchmark Phases

This diagram reflects the different Iteration/phases for the Banking, Ecommerce, and Support workloads.

**Power:** Power data is collected during phase C test run time and stored for later retrieval into the results files by the harness. The data is summarized after test completion and added to the results file per run rules.

**Temperature:** Temperature data is collected during all iterations for later retrieval into



the results files by the harness.

**Phase A:** Ramp-up period is the time period across which the load generating threads are started. This phase is designed to ramp up user activity rather than beginning the benchmark run with an immediate and full-load spike in requests and sends at least one request per user thread.

**Phase B:** The warm-up period is intended to be a time during which the server can prime its cache prior to the actual measurement interval. At the end of the warm-up period, all results are cleared from the load generator, and recording starts anew. Accordingly, any errors reported prior to the beginning of the run period will not be reflected in the final results for this benchmark run.

**Phase C:** The run period is the interval during which benchmark results are recorded. The results of all HTTP requests sent and responses received during this interval will be recorded in the final benchmark results. During this phase the system power data is collected for later retrieval into the results files by the harness.

**Phase D:** The thread ramp-down period is simply the inverse of A. It is the period during which all load-generating threads are stopped. Although load generating threads are still making requests to the server during this interval, all recording of results will have stopped at the end of the run period.

**Phase E:** The ramp-down period is the time given to the client and server to return to their "unloaded" state. This is primarily intended to assure sufficient time for TCP connection clean-up before the start of the next test iteration.

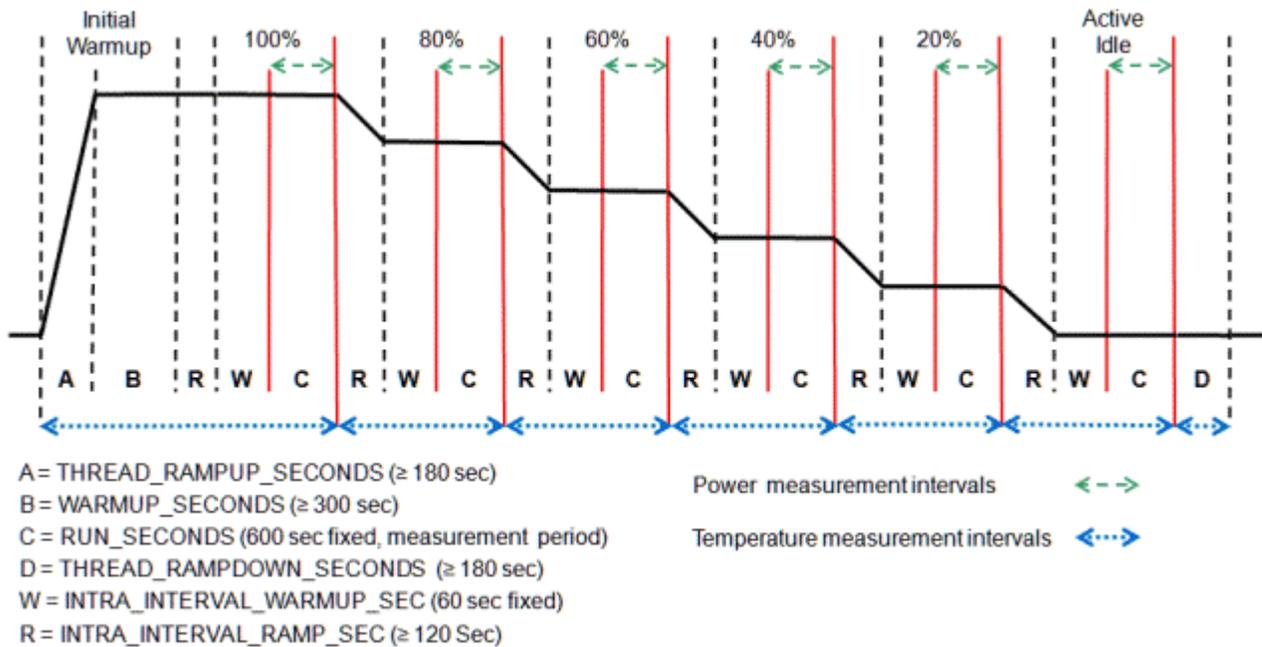
**Phase F:** The ramp-up period replaces the warm-up period, B, for the second and third benchmark run iterations. It is presumed at this point that the server's cache is already primed, so it requires a shorter period of time between the thread ramp-up period and the run period for these subsequent iterations in order to reach a steady-state condition.

The minimum intervals chosen for each of these periods are based on what was needed to get a stable run. The user may want to increase these

(`THREAD_RAMPUP_SECONDS`, `THREAD_RAMPDOWN_SECONDS`, `WARMUP_SECONDS`) to higher values when running higher loads. Other values such as `RUN_SECONDS`, `RAMPUP_SECONDS` and `RAMPDOWN_SECONDS` are fixed for repeatability and must be kept that way, at least for publishable runs.



### 2.3.2.2 Power Workload Timing



#### SPECweb209 Power Workload Benchmark Phases

This diagram reflects the different Iteration/phases for the Power workload.

**Power:** Power data is collected during phase C test run time and stored for later retrieval into the results files by the harness. The data is summarized after test completion and added to the results file per run rules.

**Temperature:** Temperature data is collected during all iterations for later retrieval into the results files by the harness.

**Phase A:** Ramp-up period is the time period across which the load generating threads are started. This phase is designed to ramp up user activity rather than beginning the benchmark run with an immediate and full-load spike in requests and sends at least one request per user thread.

**Phase B:** The warm-up period is intended to be a time during which the server can prime its cache prior to the actual measurement interval. At the end of the warm-up period, all results are cleared from the load generator, and recording starts anew. Accordingly, any errors reported prior to the beginning of the run period will not be reflected in the final results for this benchmark run.



**Phase C:** The run period is the interval during which benchmark results are recorded. The results of all HTTP requests sent and responses received during this interval will be recorded in the final benchmark results. During this phase the system power data is collected for later retrieval into the results files by the harness.

**Phase D:** The thread ramp-down period is simply the inverse of A. It is the period during which all load-generating threads are stopped. Although load generating threads are still making requests to the server during this interval, all recording of results will have stopped at the end of the run period.

**Phase R:** The `Intra_Interval_Ramp` period is used to ramp down the sessions/threads for the next iteration of the power workload.

**Phase W:** The `Intra_Interval_Warmup` period is used in place of the longer phase B warmup period for consecutive runs since there is no ramp down of threads between power workload iterations.

The minimum intervals chosen for each of these periods are based on what was needed to get a stable run. The user may want to increase these (`THREAD_RAMPUP_SECONDS`, `THREAD_RAMPDOWN_SECONDS`, `WARMUP_SECONDS`, `INTRA_INTERVAL_RAMP_SECONDS`) to higher values when running higher loads. Other values such as `RUN_SECONDS`, `RAMPUP_SECONDS`, `RAMPDOWN_SEC`, and `INTRA_INTERVAL_WARMUP_SEC` are fixed for repeatability and must be kept that way, at least for publishable runs.

### 2.3.3 Harness Software Design

The benchmark harness has three primary software components: the prime client code, the client "base" code, and the client workload code. Additionally, the reporter code is used to generate HTML and ASCII-formatted reports from the "raw" reports created by the prime client.

The SPECweb2009 harness separates the common benchmark harness functionality contained in the prime client code and the client base code from the workload-specific functionality contained in the client workload code. In designing it in this way, workloads can more easily be added, removed, or replaced without changes to the base harness code. The workload-specific client code for SPECweb2009 consists of three class files, one for each type of workload: `SPECweb_Banking`, `SPECweb_Ecommerce`, and `SPECweb_Support`. Adding or replacing a workload simply requires that the new class implement the methods expected by the base harness code, and that the workload's



base name match the base name of the corresponding workload configuration file. The key harness class files are illustrated, below.

The three classes invoked from the command line are specweb, specwebclient, and reporter.

**Specwebclient Harness:** is invoked on one or more client systems to start the client processes that will generate load against the HTTP server. Once invoked, specwebclient listens on the assigned port waiting for the prime client's instructions.

### **SPECweb2009 Client Harness Flow Overview:**

When specwebclient is first invoked, it checks the command line arguments and modifies the runtime behavior according to the parameters provided (see design document for details). It then starts an RMI server and creates a specwebclient class object to bind to the default (or specified) port to listen for commands from the prime client. At this point, the specwebclient object simply waits for the prime client to send it commands.

Note: because specwebclient's createThreads() method creates the LoadGenerator object, createThreads() must be called before:

```
setup()  
start()  
stop()  
waitComplete()  
getStatistics()  
clearStatistics()  
cleanUp()  
collectServerClose()
```

as these methods all call methods on the LoadGenerator object.

### **LoadGenerator**

For the (large) majority of the remote methods invoked on the specwebclient by the prime client, the specwebclient process simply calls an identically named method on the LoadGenerator class it created. The principle LoadGenerator methods invoked by specwebclient are:



1. **createThreads():** creates a workload-specific WorkloadScheduler (ex: BankingWorkloadScheduler) via the WorkloadDispatcher for each client process/thread emulated on the client system, assigns an ID to each scheduler, and adds each scheduler to a Vector of schedulers.
2. **start():** calls each WorkloadScheduler's start() method
3. **stop():** calls each WorkloadScheduler's stop() method
4. **waitComplete():** calls each WorkloadScheduler's waitComplete() method
5. **getStatistics():** returns the ResultSet (the set of results from all WorkloadScheduler processes/threads)
6. **clearStatistics():** clears from the ResultSet all result statistics collected up to that point (called between warm-up and run times)

As these methods suggest, LoadGenerator's primary contribution to the harness functionality is to create the workload threads. Each workload thread then controls how start(), stop(), and waitComplete() are implemented.

## WorkloadScheduler

The WorkloadScheduler class is the parent class of all workload classes. In the case of the three workload classes used in the SPECweb2005 benchmark, all three workloads use the HTTP protocol, so in order to minimize code duplication among workload classes, the shared HTTP protocol-specific code was developed as the child class of the WorkloadScheduler class, and the workload classes became child classes of the HttpRequestScheduler class.

The key methods of the WorkloadScheduler class are:

- **start():** creates a worker (workload) thread and sets the data rate for the request. It then calls makeRequest() to generate a request to send to the server. Repeats this process until test is stopped.
- **makeRequest():** an *abstract* function (each individual workload type defines how to make a request, and what a request consists of)
- **stop():** sets the stopped boolean flag to true and notifies all threads
- **waitComplete():** wait for all threads to complete the request they were executing prior to the test being stopped.



## Connection

The Connection class objects are created when the workload-specific class objects are created (i.e. in the workload-specific class' constructor). For the Banking workload, by default two SSLConnection objects are created in the SPECweb\_Banking class constructor. For Ecommerce, two SSLConnection objects and two Connection (non-SSL) objects are created to handle the two different types of requests. For Support, it creates two Connection objects.

Each of the Connection objects (whether they be SSL or non-SSL) created:

- Creates a new TCP Socket (an SSL socket in the case of an SSLConnection)
- Creates a CommReader object to read from the socket input stream (this is where the client spends a large percentage of its time reading responses from the web server to requests made)
- Creates a CommWriter object to write to the socket output stream (requests to the web server are sent via the CommWriter)
- Sets other socket parameters (KA, etc)

The two key methods in the Connection class are sendRequest() and readResponse(). sendRequest() writes request string to the socket output stream via the CommWriter object and readResponse() conversely reads the response from the socket input stream via the CommReader object.

## BankingWorkloadScheduler's makeRequest() method

As indicated above, the workload-specific makeRequest() method is completely user-defined. The harness has no expectation of any specific functionality. It is simply going to repeatedly invoke this method until the test time has expired. So makeRequest() can be thought of as a method creating the smallest unit of work that the workload will perform.

In the case of SPECweb\_Banking, the makeRequest() method:

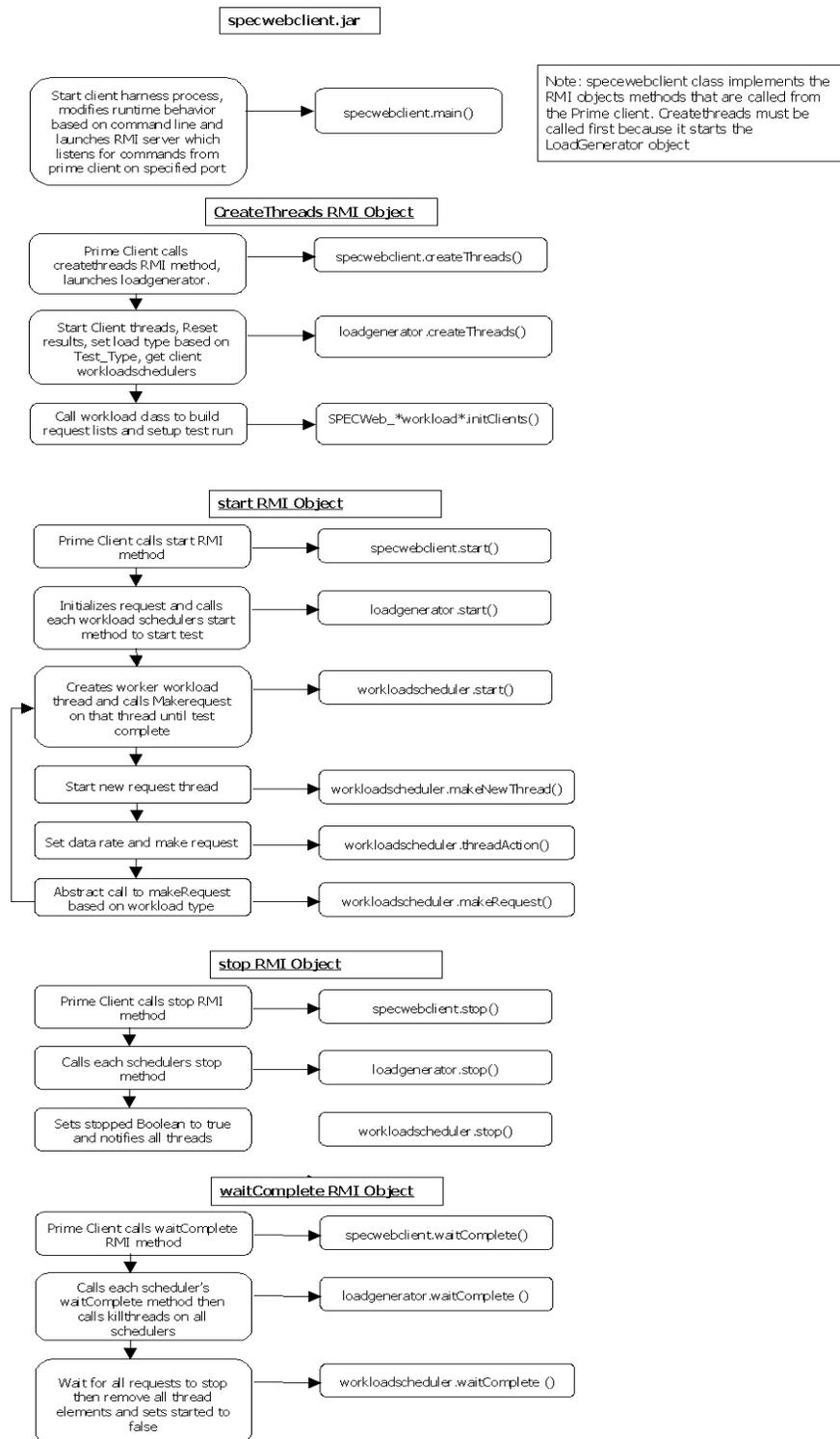
- sets the data rate for the connection (based on the rate passed in to the method)
- initializes a series of variables specific to the (randomly chosen) account numbers being accessed, including which type of request is going to be made
- builds the request string specific to the type of request being made
- sleeps for the required think time interval (via the thinkFirst() method)
  - **remainder = tmp - (now - lastStartTime - pageLoadTime);**



- calls `makeHttpRequest()` passing it the request string generated
- determines based on the value returned from the `makeHttpRequest()` method whether the request returned successfully. If unsuccessful, it adds an "error" result to the result set. If successful, it then validates the result returned.
- If validation is successful, it then requests the page's supporting image files via the `makeSimGetRequests()` method. If all of the page image files are returned and validated successfully, then a successful result is added to the result set. Otherwise, an appropriate error result is added to the result set.
- Checks to see if the server sent a "Connection: close" in the header. If so, it closes the sockets on the client side.
- And finally, it checks to see if this was the final request of the user session, and if so, closes the sockets on the client side.



## SPECweb2009 Client Harness Flow Diagram:





**Prime Client Harness:** is started by invoking specweb on the prime client system. specweb reads in the relevant configuration files (Test.config, Testbed.cong, and the workload-specific configuration file) and then lets the SPECwebControl class handle benchmark execution. SPECwebControl then creates a RemoteLoadGenerator to handle communication between the prime client and the client processes (represented by the blue line, above). RemoteLoadGenerator communicates with the specwebclient processes via Java's Remote Method Invocation (RMI).

## **SPECweb 2009 Prime Client Process Overview:**

### **SPECweb**

When specweb is first invoked, it checks the command line arguments and modifies the runtime behavior according to the parameters provided ( see design document for details). It then creates an instance of the specweb class that reads the configuration files and creates the Configuration class object, creates a Validator object with which it then changes any non-compliant configuration values back to complaint values, if so configured, and then creates and instantiates the GUI, if used.

The method startBenchmark() is then invoked on the specweb class object. This method validates the configuration and creates the SPECwebControl class object that (descriptively) assumes control of the remainder of the benchmark run activity on the prime client (controller) side.

### **SPECwebControl**

SPECwebControl's start() method then creates a dispatch thread that calls doBenchmark(), which runs remainder of the test.

The runPowerWorkload or runOriginalWorkload method:

- Gets workload information from Configuration object and sets number of connections per client system (setWorkload())
- Creates a RemoteLoadGenerator object for Remote Method Invocation on each client
- Uses RMI to call createTheads()(RemoteLoadGenerator.createThreads())
- Uses RMI to call start() (RemoteLoadGenerator.start())
- Sleeps through benchmark warm-up
- Clears any returned run statistics
- Starts benchmark run (pollClients())
- Uses RMI to call stop() (RemoteLoadGenerator.stop())



- Uses RMI to collect statistics from clients (`RemoteLoadGenerator.getStatistics()`)
- Reports results (`reportResults()`)

A couple of other methods of possible interest within `SPECwebControl`:

`pollClients()` basically does nothing unless you have set `POLL_CLIENTS=1` and/or `USE_GUI=1`, in which case it polls the clients for interim data at (fixed) 30-sec intervals.

`reportResults()` opens a results file, creates the `ResultRawReporter` object, and writes the configuration and raw results to the results file.

### **specwebPTDInterface**

*Main interface into the PTDdaemon code that user launches separately. Harness `specwebcontrol.doBenchmark` method launches the `setServerAddress`, `connect`, and `disconnect` methods on the class. During the benchmark run either the `runOriginalWorkload` or `runPowerWorkload` will start the PTD data collection by running the `go` method and stop data collection by running the `stop` method.*

### **RemoteLoadGenerator**

The `RemoteLoadGenerator` class handles all communication on the prime client side with the remote clients (i.e. RMI happens here). There is one `RemoteLoadGenerator` object per workload type. (Since the current benchmark only runs a single workload at a time, this effectively means there is one `RemoteLoadGenerator` class object per benchmark run.)

The `RemoteLoadGenerator` class object communicates with each of the remote clients via an array of `RemoteControl` objects called `remoteClients`. In order to be able to communicate with these remote clients in parallel rather than serial, the `RemoteLoadGenerator` class creates and communicates with these remote clients on separate threads.

The following remote methods are called by `RemoteLoadGenerator` on these (parallel) threads:

- `createThreads()`
- `start()`
- `stop()`
- `clearStatistics()`
- `waitComplete()`

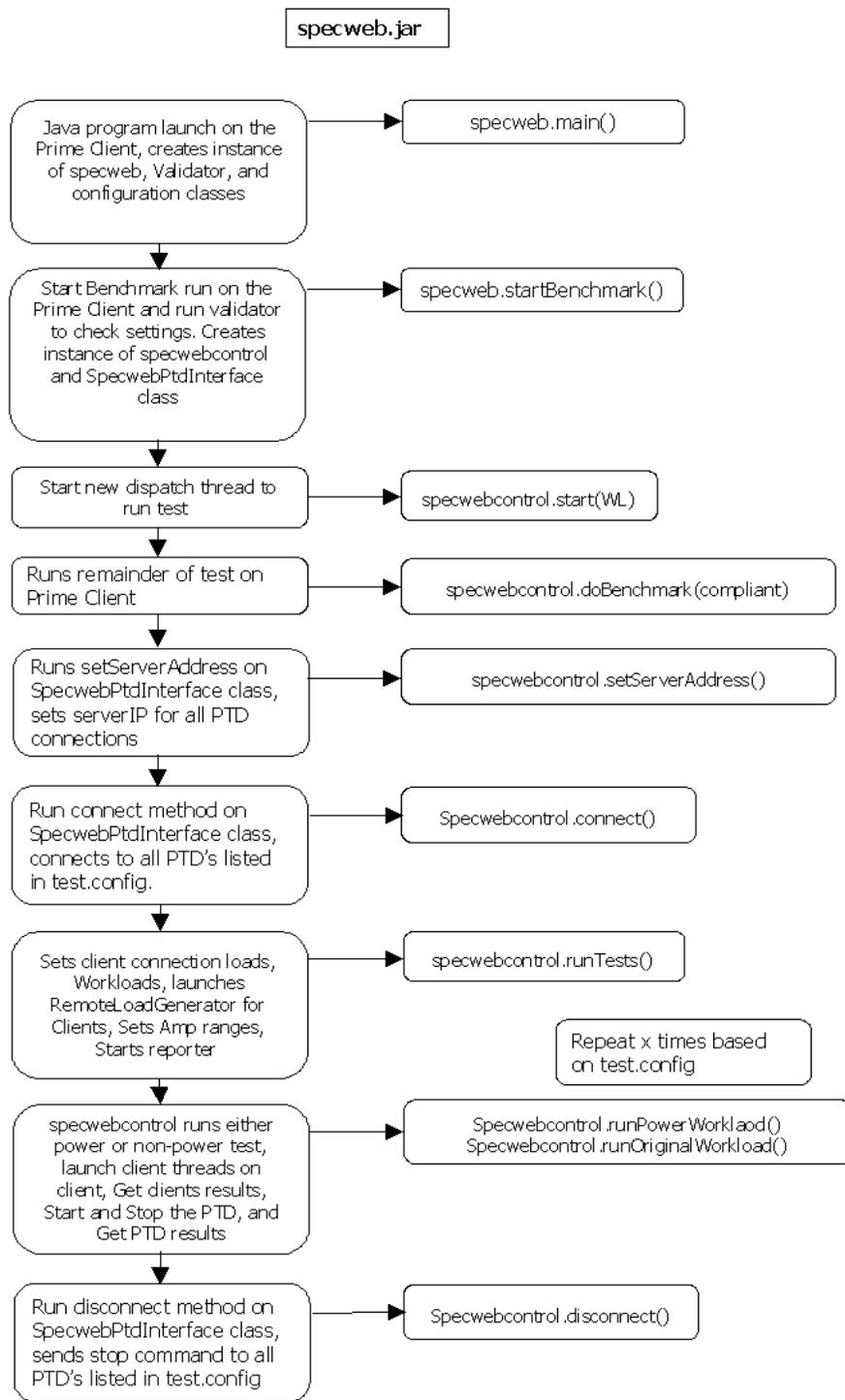


- `collectServerClose()`

Conversely, the following remote methods are not called on these parallel threads:

- `cleanUp()`
- `getStatistics()`
- `getHeartbeat()`
- `exit()`

Of the four remote methods immediately above, with the exception of `getStatistics()`, these could potentially also be called on the parallel threads, though not doing so in no way compromises the accuracy of the benchmark results. However, with the exception of `collectServerClose()`, the five other remote methods called on parallel threads must happen in parallel. For the `getStatistics()` remote method, the decision to exclude it from being called in parallel was based on concerns over how well the prime client could handle a potentially very large amount of aggregate result data being sent from all of the clients at the same time.



The reporter class is invoked only to *recreate* the ASCII and HTML results for a



previously created single or combined raw file, or to combine 3 separate workload .raw files into one submittable .raw file for SPEC submission. The latter is the method to initially generate the raw and formatted files for a complete SPECweb2009 result. Note that the individual .raw files created at the end of the benchmark run for each of the workloads are essentially "interim" results files. They must be combined using the reporter to create a submittable raw file for SPEC submission.

### **2.3.4 Rated Receive and Quality of Service**

SPECweb2009 uses a rated receive mechanism for simulating connection speeds. The simulated connection speeds have a maximum rate of 100,000 bytes/sec. This rate was chosen to reflect the higher connection speeds common today with the widespread adoption of broadband.

SPECweb2009 uses a web page-based QOS. For all except the Support workload's "download" request, a time-based QOS is used. Specifically, QOS is based on the amount of time that elapses between a web page request and the receipt of the complete web page, including any supporting image files. For the Support workload's download state, a more appropriate byte rate-based QOS is applied. That is, the total bytes received from a download request divided by the download time must meet the byte rate stipulated in the QOS requirements.

### **2.3.5 Think Time and HTTP 304 Server Responses**

SPECweb2009 includes a "think time" between user session requests (i.e. between workload "states"). After the initial request in a user session, the load generating thread will calculate a random, exponentially-distributed think time between the values of THINK\_INTERVAL and THINK\_MAX, with an average value of THINK\_TIME, as specified in each workload's configuration file.

The load generating thread will then wait that amount of time before issuing another request for that session. This delay is meant to more closely emulate end-user behavior between requests. In doing so, connections to the server are kept open much longer than they would, otherwise, and benchmark tuning requires more judicious choices for the server's keep-alive timeout value (particularly for SSL connections), as is the case for real-world web servers.

SPECweb2009 supports server 304 responses (not-modified-since). This is accomplished by calling the HTTP server's init script, which returns the current time on



the server. The harness then uses that value as the time value in all subsequent "if-modified-since" requests to the HTTP server, assuring that the server will return an HTTP 304 response. How frequently each static image request results in a 304 response is controlled via the "304 response %" value assigned to each of the static image files in the workload-specific configuration file.

## 2.3.6 Workload Design

Workload-specific design documentation can be found using the following links:

1. SPECweb Banking  
<http://www.spec.org/web2009/docs/design/BankingDesign.html>
2. SPECweb Ecommerce  
<http://www.spec.org/web2009/docs/design/EcommerceDesign.html>
3. SPECweb Support  
<http://www.spec.org/web2009/docs/design/SupportDesign.html>

There is no specific documentation for the SPECweb\_Power workload, because the Power workload is essentially the Ecommerce workload run at varying load levels.

## 2.3.7 Harness Configuration Files

The benchmark kit includes four workload specific configuration files. In each configuration file, variables are separated into "configurable" and "do not modify" categories.

In addition to making it easier to create and add new workloads to the SPECweb2009 harness, the harness' flexibility has been greatly increased by making it more easily modified via the configuration files. For example, state transitions can be modified by changing the state transition probabilities in the workload-specific files (the STATE\_'n' values). The static image files that are requested along with the dynamic web page can be modified by changing the PAGE\_n\_FILES values, also in the workload-specific configuration file. You can even change the names of the image files requested, their sizes, and the frequency with which requests for these image files return a 304 response (not-modified-since), simply through changes to the Image File Details section of the same configuration file.



Obviously, such changes would result in non-compliant benchmark runs, but adding such flexibility increases the benchmark's usefulness in capacity planning and other research.

## **2.4 Conclusion**

SPECweb2009 represents a standardized benchmark for measuring web server performance and power efficiency. Building upon the success of its predecessors, SPECweb2009 provides users an objective measure allowing users to make fair comparisons between results from a wide range of systems.

SPECweb2009 includes four separate workloads representing three common yet disparate types of consumer activity: shopping, banking, and downloading files. (Note that even though we have four workloads, Power is the same as Ecommerce). By offering PHP and JSP scripts to generate the dynamic server responses, SPECweb2009 also assures that the HTTP server's dynamic request performance is based on two of the most common technologies for generating dynamic content today. And though SPECweb2009 is not designed as a capacity planning tool, by exposing more benchmark tuning flexibility in the configuration files and making it easier to add custom workloads, SPECweb2009 further extends its potential usefulness in capacity planning.

This document has described the benchmark architecture. It is not a guide to running the benchmark. For information on running the benchmark please refer to the User Guide included with the benchmark CD. Also, please refer to the Run Rules that govern what constitutes a valid SPECweb2009 run prior to running tests whose results will be submitted to SPEC for publication on the SPEC web site or publicly disclosed as a valid SPECweb2009 result.

## **2.5 Additional Information**

More information on SPECweb2009 can be found at the SPEC web site at:

<http://www.spec.org/web2009/>



### 3. System Configuration

System Under Test (SUT)	
# of SUTs	1
Vendor	Hewlett-Packard
Model	HP ProLiant DL370 G6
Processor	Intel Xeon W5580
Processor Speed (MHz)	3200
# Processors	8 cores, 2 chips, 4 cores/chip, 16 threads
Primary Cache	32 KB I + 32 KB D on chip per core
Secondary Cache	256 KB I+D on chip per chip
Other Cache	8 MB I+D on chip per chip
Memory	48 GB (6x8GB PC3-10600R DDR3)
Disk Subsystem	1x146GB SFF SAS 15K RPM, 12x120GB SFF SATA SSD
Disk Controllers	Smart Array P410i (embedded), Smart Array P410 Controller with 512MB cache
Operating System	RedHat Enterprise Linux 5.3 (2.6.18-128.el5)
File System	ext2
Other Hardware	2xHP ProCurve Switch 3400cl-48G (J4906A), 2xHP Procurve Media Flex Module (J8435A), 4x HP Procurve SR Transceiver (J8436A)
Other Software	Java(TM) SE Runtime Environment (build 1.6.0_14-p-b03), Java HotSpot(TM) 64-Bit Server VM (build 15.0-b07, mixed mode)



<b>SUT Network</b>	
# of Controllers	4
Network Controllers	2x ServerEngines Raywire2 Dual-Port 10 GbE SFP+ Converged Network Adapter (BE4TGX14-P01) 2xNC382i (embedded dual port)
# of Networks	7
Network Type	10 Gigabit 1 Gigabit Ethernet
Network Speed	10 Gb/s 1 Gb/s
MSL (sec)	30 (Non RFC1122) 30 (Non RFC1122)
Time-Wait (sec)	60 (Non RFC1122) 60 (Non RFC1122)
MTU Size	1500 1500

<b>Web Server (HTTP) Software</b>	
Vendor	Accoria Networks, Inc.
Name/Version	Rock Web Server v1.4.7 (x86_64)
Dynamic Scripts	JSP
Server Cache	N/A
Log Mode	Common Log Format

<b>JSP Script Engine</b>	
Vendor	Accoria Networks, Inc.
Name/Version	Rock JSP/Servlet Container v1.3.2 (x86_64)
Dynamic Scripts	JSP
Server Cache	N/A
Log Mode	Common Log Format



<b>Clients</b>	
# of Clients	64
Model	HP ProLiant DL360 G3
Processor	Intel Xeon
Processor Speed (MHz)	3060
# Processors	2 cores, 2 chips, 1 core/chip, 4 threads (Hyper-Threading Enabled)
Memory	4 GB
Network Controller	NC7170 (embedded)
Operating System	Windows Server 2003 SP1
JVM Version	Java 2 Runtime Environment, Standard Edition (build 1.5.0_06-b06)
JIT Version	Java HotSpot Client VM (build 1.5.0_06-b05, mixed mode, sharing)
Other Hardware	HP ProCurve 2848 for client management on second embedded NIC
Other Software	N/A

<b>Backend Simulator (BESIM)</b>	
# of Simulators	3
Model	HP ProLiant DL385
Processor	AMD Opteron 275
Processor Speed (MHz)	2200
# of Processors	4 cores, 2 chips, 2 cores/chip, 4 threads
Memory	8 GB
Network Controller	HP NC7781 PCI-X Gigabit embedded
Operating System	RHEL 4 U4 (kernel 2.6.9-42.ELsmp)
File System	ext2
Web Server	Accoria Rock Web Server 1.4.0 (x86_64)
Server Scripts	ISAPI
Other Hardware	N/A
Other Software	N/A



## 4. SPECweb2005 Performance Results

<b>Banking</b>									
Simultaneous User Sessions	Test Iteration	Minimum QOS Compliance			Validation Errors	Average Active Power (W)			SUS/Watt
		Good	Tolerable	Fail		Server	Storage	Sum	
108096	1	95.7	99.4	0.62	0	419	0.00	419	258
	2	95.0	99.0	0.98	0	415	0.00	415	261
	3	95.7	99.9	0.06	0	425	0.00	425	255
<b>Average</b>						419	0.00	419	258
<b>Alterable / Modifiable Configuration</b>	<b>Used Components</b>					<b>Unused Components</b>			
	12x120GB SFF SATA SSD (RAID0,ext2)					None			

<b>Ecommerce</b>									
Simultaneous User Sessions	Test Iteration	Minimum QOS Compliance			Validation Errors	Average Active Power (W)			SUS/Watt
		Good	Tolerable	Fail		Server	Storage	Sum	
139200	1	98.9	100.0	0.00	0	431	0.00	431	323
	2	98.1	99.7	0.33	0	430	0.00	430	324
	3	95.7	99.0	0.97	0	431	0.00	431	323
<b>Average</b>						430	0.00	430	323
<b>Alterable / Modifiable Configuration</b>	<b>Used Components</b>					<b>Unused Components</b>			
	4x120GB SFF SATA SSD (RAID0,ext2)					4x120GB SFF SATA SSD (RAID0,ext2) mounted on /mnt/spare1, 4x120GB SFF SATA SSD (RAID0,ext2) mounted on /mnt/spare2			



<b>Support</b>									
Simultaneous User Sessions	Test Iteration	Minimum QOS Compliance			Validation Errors	Average Active Power (W)			SUS/Watt
		Good	Tolerable	Fail		Server	Storage	Sum	
66560	1	99.7	100.0	0.00	0	379	0.00	379	176
	2	99.7	100.0	0.01	0	379	0.00	379	176
	3	99.7	100.0	0.00	0	379	0.00	379	176
<b>Average</b>						379	0.00	379	176
<b>Alterable / Modifiable Configuration</b>	<b>Used Components</b>					<b>Unused Components</b>			
	12x120GB SFF SATA SSD (RAID0,ext2)					None			

<b>Power/Performance</b>						
Iteration	Performance		Average Active Power			Performance to Power Ratios
	Target Load	Simulated User Sessions	Server	Storage	Sum	
0	100%	139200	426	0.00	426	327
	80%	111360	404	0.00	404	276
	60%	83520	339	0.00	339	247
	40%	55680	292	0.00	292	191
	20%	27840	270	0.00	270	103
	0%	0	242	0.00	242	0.00
	<b>Total Sessions/Total Power =</b>					





## 5. References

1. SPECweb2009 Release 1.0 Benchmark Design Document

[http://www.spec.org/web2009/docs/design/SPECweb2009\\_Design.html](http://www.spec.org/web2009/docs/design/SPECweb2009_Design.html)

2. SPECweb2009 Results

<http://www.spec.org/web2009/results/>

3. SPECweb2009 Release 1.0 Run and Reporting Rules

<http://www.spec.org/web2009/docs/runrules.html>

4. SPECweb2009 Release 1.0 User's Guide

<http://www.spec.org/web2009/docs/usersguide.html>



# Appendix: Positioning Industry Standard Benchmarks

