MICROSOFT

Response to IBM's Whitepaper Entitled Benchmarking and Beating Microsoft .NET 3.5 with WebSphere 7

7/2/2009

This document is a response to an IBM Benchmark Rebuttal Document to our original benchmark results published at http://msdn.microsoft.com/stocktrader. Microsoft stands behind the original test results published there. This document is the Microsoft response, point-for-point, to the IBM response to our original results, and includes new benchmark data as further evidence.

Contents

Executive Summary 4

Introduction 6

The Microsoft Findings 8

Response to IBM Rebuttal 9

- IBM Friendly Bank Benchmark 9
- Microsoft Comments on IBM Friendly Bank Rebuttal Benchmark 9
- IBM CPO StockTrader/.NET Benchmark 11
- Microsoft Comments on IBM CPO StockTrader Rebuttal Benchmark 12

Summary 19

Appendix A: Microsoft .NET StockTrader and WebSphere Trade 7 Test Results for IBM's Revised Test Script 20

- Hardware Tested 20
  - Application Server Hardware 20
  - Database Server Hardware 20
- Methodology and Scripts 21
- Testing Buys and Sells 21
- .NET Buy LoadRunner Script 25
- Benchmark Results 29
- .NET StockTrader LoadRunner Summary 30
- WebSphere 7 Trade LoadRunner Summary 31
- .NET StockTrader HTTP Responses/Second 32
- IBM WebSphere 7 HTTP Responses/Second 33
- .NET StockTrader Pass/Fail Transactions per Second 34
- IBM WebSphere 7 Trade Pass/Fail Transactions per Second 35
- .NET StockTrader Transaction Response Times 36
- IBM WebSphere Trade 7 Transaction Response Times 37
- .NET StockTrader Application Server CPU Utilization during Test Run 38
- IBM WebSphere 7 Trade Application Server CPU Utilization during Test Run 39
- .NET StockTrader Application Server LoadRunner Summary on Completion of Test 40
- IBM WebSphere 7 Trade Application LoadRunner Summary on Completion of Test 41

Appendix B: Web Service Tests using no HTTP Server 42

Application Server Hardware 42
.NET WSTest EchoList Front Ended with IIS 7 vs. IBM WebSphere 7 WSTest EchoList Front Ended with IBM HTTP Server 43
.NET WSTest EchoList without IIS 7 (self-hosted WCF HTTP Service) vs. IBM WebSphere 7 WSTest EchoList without IBM HTTP Server (in process WebSphere port 9080) 44
WebSphere Tuning – Windows Server 2008 45
IBM HTTP Server Tuning Windows Server 2008 45
.NET Tuning 46

Executive Summary

In late April of 2009, Microsoft released a comprehensive benchmark report entitled *Benchmarking IBM WebSphere 7 on IBM Power6 and AIX vs. Microsoft .NET on Hewlett Packard BladeSystem and Windows Server 2008.* Recently, IBM has circulated a rebuttal document within many enterprise accounts, and that rebuttal is entitled *Benchmarking and Beating Microsoft .NET 3.5 with WebSphere 7*. The IBM rebuttal document was created by the IBM SWG CPO Performance Team at IBM. This document is the Microsoft response to that rebuttal document.

IBM's rebuttal document centers around two new benchmarks, and makes several false claims, as noted below. We stand behind all of our original findings, and point-for-point respond to IBM's claims.

- IBM did not publish any Java or .NET source code to their Friendly Bank or CPO StockTrader rebuttal benchmark workloads. Microsoft follows a full disclosure policy and publishes all source code and full testing details. This information is available at http://msdn.microsoft.com/stocktrader. IBM needs to publish all source code for their counter benchmarks.
- IBM's Friendly Bank benchmark uses an obsolete .NET Framework 1.1 application that includes technologies such as DCOM that have been obsolete for many years. This benchmark should be fully discounted until Microsoft has the chance to review the code and update it for .NET 3.5, with newer technologies for ASP.NET, transactions, and Windows Communication Foundation (WCF) TCP/IP binary remoting (which replaced DCOM as the preferred remoting technology).
- IBM makes several false claims about the .NET StockTrader:
    - **IBM claim**: The .NET StockTrader does not faithfully reproduce the IBM Trade application functionality.
      Microsoft response: this claim is false; the .NET StockTrader 2.04 faithfully reproduces the IBM WebSphere Trade application (using standard .NET Framework technologies and coding practices), and can be used for fair benchmark comparisons between .NET 3.5 and IBM WebSphere 7.
    - **IBM claim**: The .NET StockTrader uses client-side script to shift processing from the server to the client.
      Microsoft response: this claim is false, there is no client-side scripting in the .NET StockTrader application.
    - **IBM claim**: The .NET StockTrader uses proprietary SQL.
      Microsoft response: the .NET StockTrader uses typical SQL statements coded for SQL Server and/or Oracle; and provides a data access layer for both. The IBM WebSphere 7 Trade application similarly uses JDBC queries coded for DB2 and/or Oracle. Neither implementation uses stored procedures or functions; all business logic runs in the application server. Simple pre-prepared SQL statements are used in both applications.
    - **IBM claim**: The .NET StockTrader is not programmed as a universally accessible, thin-client Web application. Hence it runs only on IE, not in Firefox or other browsers.
      Microsoft response: In reality, the .NET StockTrader Web tier is programmed as a universally accessible, pure thin client Web application. However, a simple issue in the

use of HTML comment tags causes issues in Firefox; these comment tags are being updated to allow the ASP.NET application to properly render in any industry standard browser, including Firefox.

- **IBM claim**: The .NET StockTrader has errors under load.
  Microsoft response: This is false, and this document includes further benchmark tests and Mercury LoadRunner details proving this IBM claim to be false.

Introduction

In late April of 2009, Microsoft released a comprehensive benchmark report entitled *Benchmarking IBM WebSphere 7 on IBM Power6 and AIX vs. Microsoft .NET on Hewlett Packard BladeSystem and Windows Server 2008.* Recently, IBM has circulated a rebuttal document within enterprise accounts, and that rebuttal is entitled *Benchmarking and Beating Microsoft .NET 3.5 with WebSphere 7*. The IBM rebuttal document was created by the IBM SWG CPO Performance Team at IBM. This document is the Microsoft response to that rebuttal document. Ultimately, vendor competition around middle tier software performance and pricing is healthy for customers, and we believe ongoing exploration of application server performance and pricing is a key part of that competition.

It is important to note that in all Microsoft-driven middle-tier application server benchmarks involving Microsoft .NET and IBM WebSphere, a policy of full disclosure is followed; as is required in both the IBM WebSphere End-User License Agreement and the Microsoft .NET End-User License Agreement (see "Benchmarking" Clause in the respective EULAs). Full disclosure is extremely important, as it allows customers and competing vendors to fully analyze the results, and even replicate the testing on their own such that fully informed responses can be made. Full disclosure was adhered to in the original Microsoft benchmark entitled *Benchmarking IBM WebSphere 7 on IBM Power6 and AIX vs. Microsoft .NET on Hewlett Packard BladeSystem and Windows Server 2008.* This means that along with the benchmark results as documented in the paper, any customer or competing vendor is able to download at a publicly posted Web site the following materials:

- All source code used in the benchmark workloads (both the Java and .NET implementations)
- Detailed benchmark results
- Details of the benchmarking software used, and test script flow(s) used for all workloads
- Breakout of all software and software versions used in the tests
- Detail on the precise hardware used in the tests
- Charts showing the test bed setup, including network diagrams, number of clients
- Precise testing methodology (think times, number of clients, measurement periods, etc.)
- The tuning parameters/settings used for both .NET and IBM WebSphere
- Breakout and precise calculations/sources for any pricing detail when pricing data is included with the benchmark results

Such disclosure is necessary not only for fairness (such that competing vendors can examine and comment on the tests); but also for customer credibility. We encourage customers intrigued by the original Microsoft test results to download the benchmark kit (the Microsoft .NET StockTrader download); and perform their own analysis or even replication of the tests conducted so that they can verify the results for themselves. In addition, the Microsoft .NET StockTrader download (which includes all .NET and Java sources tested) includes a turnkey multi-agent benchmark tool with source code (the Capacity Planner) that is designed to make it easy for customers to perform their own analysis against the Web Service workloads included in the download. And the application itself represents a best-practice, performance-driven .NET server-based application that can be run in a variety of modes as a learning sample. Finally, the .NET StockTrader also illustrates bi-directional interoperability between

.NET and Java, based on industry standards for Web Services including SOAP and WS-* protocols. The Microsoft benchmark report and .NET StockTrader download (with all .NET and Java sources) can be downloaded from http://msdn.microsoft.com/stocktrader.

In IBM's response document, entitled *Benchmarking and Beating Microsoft .NET 3.5 with WebSphere 7*, IBM did not follow full disclosure. Their paper references two benchmark applications they tested (CPO StockTrader and Friendly Bank). However, **they have not published source code** for either benchmark workload (they published neither the Java or .NET implementations). Furthermore, IBM does not fully disclose the benchmark test bed setup with appropriate network diagrams, database load and other details required to fully analyze and comment on their findings. Without the source code to the IBM applications, customers should immediately call into question the credibility of the IBM results. Nevertheless, IBM does provide enough detail on their key rebuttal points such that we can respond to these claims to the best of our ability without the source code. That response is the focus of this paper.

In summary, we stand by our previously published results and continue to challenge IBM to **meet us in an independent lab to perform additional testing of the .NET StockTrader and WSTest benchmark workloads and pricing analysis of the middle tier application servers tested in our benchmark report. In addition, we invite the IBM competitive response team to our lab in Redmond, for discussion and additional testing in their presence and under their review.**

The Microsoft Findings

The findings of the original Microsoft benchmark study (and all .NET and Java source code and testing details) are available at http://msdn.microsoft.com/stocktrader. Customers should read this full report, as we stand behind these published results. A summary of these results from April, 2009 is included below. The complete paper at the address above includes full details including pricing breakouts of hardware and software as tested.

Platforms Tested

- **IBM Power 570 with IBM WebSphere 7 and AIX 5.3**
- **Hewlett Packard BladeSystem C7000 with IBM WebSphere 7 and Microsoft Windows Server 2008**
- **Hewlett Packard BladeSystem C7000 with Microsoft .NET and Windows Server 2008**

	IBM Power 570 with WebSphere 7 and AIX 5.3	Hewlett Packard BladeSystem C7000 with WebSphere 7 and Windows Server 2008	Hewlett Packard BladeSystem C7000 with Microsoft .NET and Windows Server 2008
Total Hardware + Operating System Costs	$215,728.08	$50,161.00	$50,161.00
Middle Tier Application Server Licensing Costs	$44,400.00	$37,000.00	$0.00
Total Middle Tier System Cost as Tested	**$260,128.08**	**$87,161.00**	**$50,161.00**
Trade Web Application Benchmark Sustained Peak TPS	8,016 transactions per second	11,004 transactions per second	12,576 transactions per second
Trade Middle Tier Web Service Benchmark Sustained Peak TPS	10,571 transactions per second	14,468 transactions per second	22,262 transactions per second
WSTest EchoList Test Sustained Peak TPS	10,536 transactions per second	15,973 transactions per second	22,291 transactions per second

Response to IBM Rebuttal

The IBM rebuttal is centered on two benchmark workloads IBM created but has not publicly published. So the first point is that IBM should immediately publicly publish their source code, as tested, for both the Java and .NET workloads they tested. Without this public disclosure, customers should question the credibility of their results, especially considering our specific technical counter-points included in this paper. In response to our original findings, IBM quotes results from the following two new benchmark applications:

1. Friendly Bank (no Java or .NET code publicly published)
2. CPO StockTrader (no Java code published, and .NET code based on an older version of .NET StockTrader)

IBM Friendly Bank Benchmark

The IBM Friendly Bank Benchmark is supposedly a reference banking application with J2EE and .NET implementations. The IBM tests include:

- Login and Logout
- Deposit/Withdraw/Transfer funds between accounts
- View Transaction History (By Account/Customer)
- View and Update Customer Profile

This looks like an interesting benchmark workload, and **we are eager to get the source code** for both Java and .NET and perform our own analysis. IBM tested this workload in two modes:

1. Running as a monolithic Web application (with logically partitioned presentation and business logic/database access running within the Web JVM/CLR instances on a single application server talking to a remote database; but no physical partitioning of the tiers).
2. Running as a distributed application, with the Web tier physically separated from the business logic/data access tier via remoting.

Microsoft Comments on IBM Friendly Bank Rebuttal Benchmark

Most importantly, IBM notes that the .NET implementation was originally coded for .NET Framework version 1.1; and no attempt was made to update it for the latest .NET Framework version 3.5 (SP1). The .NET Framework 1.1 has been out of date for over 4 years. The most significant aspects of .NET Framework 3.5 that would impact performance in tremendous ways are:

- IBM used DCOM for their .NET remoting scenarios. DCOM and in fact COM+ serviced components in general have been a legacy technology (provided for backwards compatibility only) within .NET since 2005. Performance was a key consideration for retiring DCOM, which was designed in the mid-1990's largely for now legacy VB and C++ scenarios, and not based on .NET managed code. For .NET 3.5, the latest remoting technology is Windows Communication Foundation (WCF); which offers significantly faster performance and adheres to and implements industry standards for Web Services, REST and WS-* protocols. In addition, this infrastructure

enables TCP/IP binary remoting, which is the fastest remoting technology for .NET to .NET communication, and is the closest analogous technology to Java/RMI.

- IBM should be using the newer System.Transaction Namespace for all transactions against SQL Server. This new transaction infrastructure was introduced in 2005 with .NET 2.0. It has the ability not only to perform transactions much more quickly than COM+ Serviced Components, but also to auto-promote transactions to distributed transactions using the Microsoft Distributed Transaction Coordinator only when needed.
- IBM should be using the newer ASP.NET Web Form/User control capabilities; all were updated significantly with .NET 2.0 in 2005. Newer ASP.NET constructs introduced with .NET 2.0 might easily impact performance in significant ways.
- It is unclear if IBM even ran this application in full ASP.NET 2.0 mode, or in the backwards compatible (but lower performing) .NET 1.1 worker process mode. This needs to be configured within IIS itself. In addition, beginning with Windows Server 2008, a new Integrated Pipeline mode for ASP.NET applications was introduced. The .NET application, if properly coded for the latest ASP.NET release (beginning with .NET 2.0 in 2005); should be running in this Integrated Pipeline mode under Windows Server 2008 and IIS version 7.
- IBM fails to even mention what Windows Server OS was used in their testing (Win Server 2003? Win Server 2008?); or whether the applications were running as older 32-bit or newer 64-bit compiled applications on modern server hardware.
- IBM mentions they have to use more than one worker process for the Friendly Bank .NET Web application. This alone speaks to serious issues with their implementation; a modern-day .NET ASP.NET Web application will perform, if properly coded to SQL Server, up to full CPU saturation and peak TPS using a single worker process on the Web tier.
- IBM does not even disclose, for this benchmark, what edition/version of SQL Server they used. Today, SQL Server 2008 is the current release, and they should be using the latest ADO.NET (the integrated version and .NET data access providers shipped with .NET 3.5) data access technology as we demonstrate in the .NET StockTrader application, as coded for peak performance using model classes and stateless ADO .NET DataReaders (with System.Transactions used for all database transactions in the business tier). It does not appear as if any of this was done.
- No think times were used on client load drivers; they only simulated a very limited number of client connections; and no details were provided on the test bed setup. Contrast this to how the Microsoft tests were conducted for .NET StockTrader (many physical machines, thousands of simulated users, in a much more realistic test; with full test bed details and network diagrams published).

Just based on these points alone, the Friendly Bank benchmark results as published by IBM should be fully discounted. IBM tested a .NET Framework 1.1 application, built on non-.NET and .NET technology that has been obsolete for over 4 years. We encourage IBM to send us all sources as tested for both Java and .NET. **We will bring the .NET implementation up to date, test on a realistic test bed setup, publish our .NET implementation source code and complete test results and details. Such results would likely be quite valuable for customers.**

IBM CPO StockTrader/.NET Benchmark

IBM originally published their implementation of this application as the Trade benchmark application, as designed and programmed by the WebSphere performance team. This workload has been used for many years as their standard performance-driven benchmark workload and capacity planning tool for IBM WebSphere. They have kept this application up-to-date over the years, by publishing updates to the application through IBM WebSphere 6.1; hence their latest version, publicly downloadable, is Trade 6.1, updated for IBM WebSphere version 6.1. This application is promoted by IBM as a design reference for high-performance WebSphere/Java applications, and has been benchmarked by IBM on mainframes, Power6 and x86/x64 hardware; and is deeply used in many IBM Redbooks as a reference application/workload for enterprise-scale WebSphere deployments.

For the Microsoft benchmark, two implementations of this workload were tested, with all sources published on the public Web site http://msdn.microsoft.com/stocktrader:

1. We created a .NET implementation of their application that represents a best-practice .NET reference application using the latest .NET Framework 3.5 (SP1) technologies. This application is not a direct port of the JSP/JDBC-based IBM application: a direct port of a Java application to .NET/C# would not represent a best-practice .NET application. A strict "port" of one application from one platform to another will typically perform much worse on the secondary "ported" platform, as it will not be based on best-practice design patterns for the secondary platform (it instead will be strictly a "port" using the best practice design patterns for the original platform). Rather, it is much more valuable to test a .NET implementation that is coded to best .NET practices for performance and scale. This is perfectly valid if the .NET application is a faithful replication of the original Java workload. This means the .NET implementation must have exactly the same end-user functionality and behavior with respect to transactions, database integrity, database schema, remote web service calls, data contracts, etc. **The .NET StockTrader version 2.04 is such a faithful replication to the IBM WebSphere Trade performance application, and customers can look at the code provided for both applications, and run both applications on their own.**
2. We updated the Trade 6.1 IBM Java code for IBM WebSphere 7.0 and ran it against the latest DB/2 (at the time DB2 version 9.5) with the latest IBM Java runtime and IBM DB2 JDBC drivers. The specific updates that were made to the Trade 6.1 application to bring it up to date to WebSphere 7 were:
    a. Streamlining the database access tier to remove EJB 2.0 technologies, and instead use higher performance in-process calls to a business service tier using direct JDBC access to the database based on pre-prepared SQL statements.
    b. Recompiling and running on the newer IBM JVM (6.x) with latest Java Enterprise features and deprecated features removed.
    c. Running on the latest 7.x edition of the IBM HTTP Server (Apache).
    d. Running as 64-bit applications on 64-bit WebSphere on modern server hardware (not older 32-bit hardware).

e. Re-programming the Web Service façade to use IBM's newer JAX-WS Web Service technology and data-binding stack; the older JAX RPC technology has been replaced by JAX-WS in the Java and IBM WebSphere stack, and performs better than JAX-RPC.
f. Ensuring the use of the same data and service contracts between .NET and WebSphere for all Web Service/business service functions. This is demonstrated by 100% bi-directional, seamless interoperability between the .NET and WebSphere business service and UI tiers. This interoperability itself demonstrates that behavioral functionality between the .NET and Java implementations was strictly adhered to (down to the database schema, data constructs and transactional behavior of the business service, data access and UI tiers).
g. Ensuring that the user interface HTML pages return roughly (actually still a little bit less than the more polished .NET pages) the same number of HTML bytes per request as the .NET implementation, for fair benchmark comparisons.
h. Microsoft published the full source for the updated Java/WebSphere version 7 Trade benchmark application, and this can be downloaded as part of the StockTrader download at http://msdn.microsoft.com/stocktrader for customer review and testing. We stand behind both implementations as a very fair comparison of two equivalent applications, each coded to their respective platforms best-practices for high performance.

Microsoft Comments on IBM CPO StockTrader Rebuttal Benchmark

In IBM's rebuttal paper, they make several inaccurate claims.

1. First and foremost, they claim that the .NET StockTrader 2.04 benchmark application is not a faithful reproduction of the WebSphere Trade application, but rather is a proprietary implementation that includes "little, if any, of the original design." This is false. As discussed earlier, the .NET StockTrader 2.04 implementation faithfully reproduces the IBM workload across all tiers, UI, business services and data access, but uses a .NET design pattern. However, even this design pattern follows the IBM pattern with the same database schema and database model objects used to represent and pass data between data access, business service and UI tiers. This is evident since the WebSphere JSP application can seamlessly interoperate with the .NET middle tier (via Web Service mode, with no logic changes required); and the ASP.NET Web tier can seamlessly interoperate with the WebSphere/Java middle tier (again, with no code changes required, just a URI configuration).
2. IBM makes the point that the .NET StockTrader is clearly a different application than IBM's Trade application because "it is not universally accessible web application since it can only be accessed using Internet Explorer, and not by other Web browsers." At first blush, this is true because of a mistake made in the HTML pages returned by ASP.NET; and one that should have been caught by Microsoft with full testing on Firefox, and not just IE. It turns out that HTML comment tags embedded in the ASP.NET HTML pages use the following syntax within the pages to denote different sections of the HTML (for example, headers, footers, etc):

- `<!------Begin Main Web Form Declaration -------->`
  While Internet Explorer accepts this and renders all pages appropriately, this is not in strict compliance with HTML guidelines for comment tags, which should look as follows:
- `<!-- Begin Main Web Form Declaration -->`
  Firefox fails to render .NET StockTrader pages appropriately because of this mistake we made with the comment tags, which we are working on correcting with an updated download to be made available at http://msdn.microsoft.com/stocktrader. We apologize for this mistake, and understand why the IBM team may have drawn the conclusion that the .NET StockTrader is not a universally accessible Web application and in this regard is different than the WebSphere Trade application.

  In fact, any customer (or IBM) can simply replace these comment tags in the .ASPX pages with the proper tags as shown above, and see the full functionality/layout of .NET StockTrader works great in IE and Firefox, and should in other browsers as well. **Furthermore, the comment tags in no way impact the validity of the benchmark results Microsoft published**. The benchmarking tool for the Web application workloads was Mercury LoadRunner, and the client driver simply downloads the full HTML page during test runs, but does not render the pages in an actual browser. Extreme care was taken in all benchmark runs to ensure the full pages were downloaded (byte counts taken), without errors, and we still fully stand behind our original results. However, we will correct this simple HTML comment tag issue, and apologize for this mistake since our intent is to have the .NET StockTrader Web UI accessible by any industry-standard browser.

3. Next, IBM claims that Microsoft "employed client side scripting to shift some of the application function to the client." **This is simply false, the IBM team is wrong**. No application functionality in the .NET StockTrader runs on the client; it runs on the server just like the IBM Trade application. No application functionality is running in client side script, and this will be immediately evident to any ASP.NET developer that looks at the code. In fact, in all benchmark test runs, a Mercury benchmark web client is used that does not even process client side script; so it would be literally impossible to benchmark the .NET StockTrader if it did employ any client-side scripting logic. Customers can see this for themselves by simply disabling client-side script in the browser, and the application still functions as normal. It is unclear why the IBM team thinks functionality is running in the browser via client-side scripting. It is not.

4. Next, IBM claims that in the Web Service workloads, an unnecessary HTTP Server was inserted in the IBM WebSphere test runs. In fact, in the IBM WebSphere test runs we used IBM HTTP Server front-ending the IBM WebSphere application server; and in the .NET test runs we used IIS 7 as the front-end Web Server front-ending the WCF Web Service processing. This a typical deployment, with the use of each vendor's preferred HTTP Server as the HTTP server, and in each case co-locating this server on the same machine running the application logic.

   However, it is true IBM has in-port HTTP listener (typically port 9080), which could be directly

exposed on the Internet/Intranet without the use of any HTTP Server. This is fully discussed in the original paper. This would be contrary to IBM best-practice deployment guidelines; typically this port is used from a dedicated, fully functional HTTP Server such as Apache or IIS. It is also true that IIS 7 and .NET have a much more integrated pipeline mode (that preserves process isolation in separate worker processes) and performs in general much better than IBM WebSphere's plugin approach. We agree that the benchmark for Web Services would perform better on IBM WebSphere 7 if the benchmark agents talk directly to IBM's internal http port 9080, rather than going through IBM HTTP Server (Apache). We do not agree this represent's IBM's own customer guidance for deployment; but in Appendix B of this paper, we publish results for running IBM WebSphere in this mode, without any HTTP Server. However, it should be noted that we also publish the equivalent Microsoft results using WCF in self-hosted mode, an optional mode WCF Web Services can similarly run without the use of an HTTP Server (IIS 7)—and this mode provides higher TPS rates for .NET as well.

Additionally, the .NET StockTrader download includes the Capacity Planner tool that enables customers to perform these tests with or without an HTTP Server on their own (all sources are included) to see this impact for themselves.

5. Next, IBM claims they ran the .NET StockTrader 2.0 application in their rebuttal benchmark, and received significant errors during their test runs, resulting in a loss of business when using .NET. They do not report what these errors were (they can look in the application event log if these are processing errors, since all server-side exceptions are recorded there). The current version of .NET StockTrader is version 2.04; and it is unclear why they chose to use an older version; especially since we have made some bug fixes in the implementation logic and we tested the newer 2.04 version in our original paper. However, even with the 2.0 version, the benchmark on .NET should not experience any errors even under high load, so they likely have some sort of script issue or mis-configuration. To illustrate this, Appendix A includes full test details of .NET StockTrader 2.04 and WebSphere Trade 7; including all HTTP status codes and error rates; customers should note for our replicated scripts that perform the same actions as IBM's rebuttal benchmark, there are no errors in either test run; with results recorded over a 30-minute measurement period using similar hardware as used by IBM in their response.

   It is possible that IBM did not pre-allocate enough space for the SQL Server database or transaction log (as we did for both DB2 and SQL Server). If so, the transaction timeout might need to be adjusted to 60 or 30 seconds, vs. the default of 15 seconds, as a small number of transactions might take longer than 15 seconds during these brief database auto-extensions. This is a simple configuration change (done via ConfigWeb in the Business Services tier). The same would be true of IBM DB2/WebSphere, but their Trade application is already using a 60 second transaction timeout by default, and it is unclear how much space they pre-allocated to DB2 in their benchmark. In short, in a proper configuration as shown in Appendix A, .Net StockTrader runs without any errors under heavy load; as does the IBM WebSphere 7 Trade application.

It is also possible IBM was running with a .NET worker process maximum queue length that was too low, resulting in HTTP 503 errors. Under heavy user loads beyond server saturation, the worker process queue length should be set higher to allow ASP.NET to queue requests vs. returning Server Too Busy (503) responses. However, the .NET StockTrader 2.04 application automatically installs worker process application pools with a queue length of 1000, and even in tests well beyond server saturation, we have not experienced any 503 errors.

Quite simply; there are no issues running .NET StockTrader 2.04 under full saturation load in a proper configuration and test bed; the application runs great and is as error-free as the IBM WebSphere 7 implementation. We were very careful to monitor all test runs for any errors during test runs, and default install options for the StockTrader 2.04 benchmark application should work error-free, as shown in Appendix A of this document.

6. Next, IBM claims that Microsoft "failed to properly monitor and adjust the WebSphere application server to achieve peak performance." This is also simply false. We conducted literally hundreds of test runs across both the Power6/AIX and Intel/Windows Server setups for WebSphere 7. We tested different heap sizes, many different thread settings for the Web container, different database connection pool sizes; and furthermore carefully turned off unnecessary features such as access logging and the WebSphere performance monitoring infrastructure. All of this was done over a period of many weeks, using performance monitoring tools on the network, database, and middle tier. We published the tuning settings used, and fully stand behind these settings.

   IBM notes that we did not publish database connection pool sizes; which is false; we used connection pools with 51 connections per JVM/WebSphere instance, as this was optimal, and this is published in the paper. In addition the SQL Statement cache size was set to 150; which is plenty large given there are less than 150 different SQL statements in the application to begin with. Again, we are fully confident in our tuning settings given the number of test runs and the fact we were able to drive all middle tiers to near 100% server saturation with no bottlenecks other than application server capacity.

   IBM did recommend and use one additional JVM tuning setting (-**Xgcthreads8**), however. In Appendix A you will see results that incorporate this additional tuning setting. We stand by our original results; this tuning setting had no impact on our re-test results (either negative or positive). We also tried -Xgcthreads4 (to match threads to number of cores on our system); also without any impact to our peak measured results.

7. Next, IBM claims that Microsoft used "proprietary SQL statements" in our implementation, unlike the IBM Trade application that was designed to be portable across databases. This point warrants some discussion. First, both applications use pre-prepared SQL statements, and neither uses stored procedures, as tested in the Microsoft benchmark. This is because in past

benchmarks that did use SQL Server Stored Procedures within .NET, some in the Java community felt stored procedures not only represented a performance advantage to .NET; but also in general should not be used in a properly coded application since stored procedures in the data access tier tie this tier to the database being used. Therefore, in all subsequent .NET/Java benchmarks (after year 2003, including this benchmark); Microsoft has used pre-prepared SQL Statements in benchmarks, and not used SQL Stored Procedures, to eliminate this as a point of contention. So both applications use pre-prepared SQL statements.

However, even with pre-prepared SQL statements, differences exist between different databases such as Oracle, SQL Server and IBM DB/2. For example, SQL Server has the construct of identity seeds, and Oracle does not, instead relying on Sequences to generate unique primary keys on inserts. DB/2 supports both. In general, it is almost impossible using ADO.NET or the Java JDBC API with pre-prepared SQL statements to use precisely the same SQL across all major databases and end up with a functional application, not-to-mention performance-based application. Many queries can be the same, but typically best-practice design for Oracle will not be the same for best-practice on DB2 or best-practice for SQL Server, even when using basic SQL for each database (which we did). Microsoft believes in a design pattern that places all SQL access in a separate dedicated, logically partitioned tier in the application, called the Data Access Layer (DAL). Different DALs can then easily be created that are optimized for best performance and full functionality on different databases. This enables the UI tier and the entire business service tier to work without any changes when the backend database implementation is changed from one database to another; yet still allow optimized performance and functionality for whatever database the customer chooses. This is how .NET StockTrader is designed. It would be literally impossible to code the .NET StockTrader for SQL Server, Oracle and DB2 using the exact same SQL statements. Interestingly, the same is true for Java and IBM WebSphere. When using the Java JDBC API (which today most high-performance applications in the Java world employ vs. the use of Container Managed Persistence), the same (usually minor) differences in SQL must be adhered to depending on the database being used for the deployment. In IBM's Trade 6.1 application, for example, there are conditional statements embedded in the data access tier that use different queries depending on whether the database supports certain SQL/EJB ordering or not. In the WebSphere 7 version of Trade (which we optimized for IBM WebSphere 7 and DB2); the same is true; we use, for example, DB2 sequences in the DB2 data access layer; and Oracle sequences in the Oracle data access implementation. The syntax between these two databases with respect to Sequences is simply different, and the JDBC queries must be different or else the application will not work on both databases (unless you use an underlying ORM technology/EJBs; however we chose to use the direct JDBC API approach since in all testing this proved to be faster than the EJB 3.x/JPA approach).

So, in a nutshell, we tested the fastest IBM Trade implementation against IBM DB2 using IBM's SQL statements as incorporated by the IBM WebSphere performance team; and the fastest .NET implementation against SQL Server 2008; each data access layer is appropriately coded to the

respective database. We continue to stand behind our results; and customers can examine the published implementations for themselves.

In addition, Microsoft in general does not believe data access tiers should be coded to lowest common denominator database capabilities. Customers choose their database based on performance and functionality and cost. It seems silly that customers would choose to build enterprise class applications without taking advantage of vendor-specific features they have paid for. It is very straightforward, as illustrated in both the .NET StockTrader (with DALs for Oracle and SQL Server); and the Trade 7 Java benchmark application (with some conditional SQL based on whether using Oracle or DB2); to isolate all data access in a streamlined data access layer; and build an optimized data access layer for the database being used, allowing the database to be easily changed with a simple configuration setting and no changes to UI or business service logic. This is largely a philosophical debate between .NET and Java architects; with .NET focused on interoperability and optimized performance; and Java largely centered on strict cross-platform and lowest common denominator implementations. We believe the .NET StockTrader 2.04 application and the IBM Trade 7 application each employ typical SQL that would be used with SQL Server (.NET); and DB2 (Java Trade 7) respectively; given that the IBM performance team themselves constructed the JDBC-based SQL statements for DB2 in their Trade application.

With that said, however, should IBM want to publish or provide us with specific new DB2 SQL Statements (we used their original SQL as the IBM Performance Team coded for Trade 6.1) using the JDBC API, we would be happy to re-test their application using these DB2 SQL statements. If they want to use DB2 packages and DB2 stored procedures; then they should publish this data access code, and let us similarly use equivalent SQL Server stored procedures in the .NET StockTrader.

Instead, they have seemingly created a completely new data access tier, and published no source code or details for this data access tier. Based on their document, which discusses the use of DB2 packages, it appears they have opted to place all their SQL in DB2 stored procedures; perhaps even with business logic; while .NET remains true to keeping business logic in a separate tier and using pre-prepared SQL statements against SQL Server.

In addition, we find it odd they tested their unpublished CPO StockTrader application using DB2 8.2, and not the latest DB2 9.7 release. Also, they used SQL Server 2005, and not SQL Server 2008 in their testing, even though SQL Server 2008 was used in the original Microsoft testing and has been available for about one year.

8. Next, IBM shows TPS results (using a revised test script and their unpublished Java code) in their rebuttal paper that simply do not make sense. They claim to have tested on an 8-proc server with a dedicated 8-proc database. They show results for WebSphere of 161 transactions per second, and .NET with 118 transactions per second. These were supposedly taken using a script

that performs the following user actions:

a. Login
b. Buy
c. Sell
d. Portfolio
e. Logout

However, on such a setup we are able to easily produce over 2,800 TPS on IBM WebSphere using just a 4-core server and 8-core database! And we report similar results for .NET for this data-driven Web application workload. It is unclear what IBM uses as their definition of a transaction; which again speaks to the incomplete nature of their paper. If they define a transaction as a complete load of the specific user pages (a complete user response) listed above, as we do; then these results they quote are impossible to understand in that they are so poor for both WebSphere and .NET. It is possible they defined a transaction as a series of pages/user responses grouped together (at a less granular level than an individual page returned to the user); if so this also seems odd and should be disclosed; because our results are simply much better on both platforms—nowhere near as low as their quoted results. This also calls into question the quality and credibility of their rebuttal benchmark.

9. Note that IBM does not comment on the pricing data included in our original report, and broken out in detail in our original report. The pricing data, in conjunction with the performance data, is what makes this benchmark so interesting. Why pay much more for a system that delivers less performance? They never comment or bring up pricing in their response. We stand behind the relative pricing data and the detail breakout we provide in the original report; as we stand behind the full performance results as published.

Summary

Microsoft stands behind the original results as published at http://msdn.microsoft.com/stocktrader. This paper provides responses to each of IBM's counter claims, and includes additional benchmark information based on IBM's counter-points. As shown in the original benchmark paper and with all source code fully disclosed; IBM WebSphere on Windows significantly outperforms the equivalent workloads as tested on the IBM Power 570/Power6 server; and at a fraction of the overall cost (hardware and software for the middle tier). In addition, the .NET results on Windows perform the best, beating IBM WebSphere on Power6; IBM WebSphere on Intel; and at a fraction of the cost of the least expensive WebSphere setup tested.

Furthermore, we request that IBM, in the interest of fairness, publicly disclose (as Microsoft has done) all source code (Java and .NET) used in their response whitepaper.

Appendix A: Microsoft .NET StockTrader and WebSphere Trade 7 Test Results for IBM's Revised Test Script

In IBM's response document, they ran a different script than our original test script. The modified script flow included a heavier emphasis on buys and also included a sell operation. We have replicated these tests using their modified script flow, and run the benchmark on a single 4-core application server. We stand by our original results as published; these tests are based on IBM's revised script and are meant to satisfy some of these IBM rebuttal test cases as outlined in IBM's response paper. They should not be considered in any way as a change to our original results (performed on different hardware, and different test script flow); as the original results remain valid.

Hardware Tested

We have repeated that test script flow as a separate verification test here. We have used a single quad-core Hewlett Packard Blade BL460c Server, equipped just like the four HP BL460c Blades used in the original test (although here just a single blade is tested, vs. 4 clustered blades):

Application Server Hardware

1 HP ProLiant BL460c
1 Quad-core Intel Xeon E5450 CPU (3.00 GHz)
32 GB RAM
2 x 1GB NICs
Windows Server 2008 64-bit
.NET 3.5 (SP1) 64-bit
IBM WebSphere 64-bit

A single database is used, which is a dual Quad-core (8 cores total) Hewlett Packard DL380 G5 server. This server is attached to two RAID arrays, each with a dedicated controller. Each RAID array is configured with a RAID 10 configuration (for fault tolerance), and each array has 14 physical 10K drives. One volume/RAID array is used for SQL Server 2008/DB2 V9.7 logging; the other array holds the actual database file.

Database Server Hardware

1 HP ProLiant DL380 G5
2 Quad-core Intel Xeon E5355 CPUs (2.67 GHz)
64 GB RAM
2 x 1GB NICs
Windows Server 2008 64-bit
SQL Server 2008 64-bit
DB2 V9.7 64-bit

The database is loaded with 100,000 quotes; 500,000 accounts; and 500,000 orders/holdings per account. IBM does not disclosure the database they used in their response document.

Methodology and Scripts

The test bed used to drive load is the same as our original setup, with 32 physical clients running the Mercury LoadRunner agent, and script think times configured at 1 second. This is a much more realistic setup than IBM's flawed response benchmark, which used a single client configured with no think times; and running just 36 threads. In our tests, as in our original tests, we find peak throughput user loads by testing iteratively up to server saturation (when response times just begin to climb a bit); and then run the 30 minute measurement period at peak throughput (but not beyond server saturation). In all tests, we are able to push the application server to near 100% saturation, indicating no external bottlenecks and an accurate test. Just as significantly, we see zero errors for both .NET and WebSphere 7 during the test runs. IBM had reported significant error rates for .NET StockTrader, and this test proves otherwise (although they never report what the errors were).

While the results very closely mirror our original results (considering this test involves just a single blade vs. 4 blades, and a single database server), they should not be directly compared to the original results since a different script is used (with heavier emphasis on buys and sells, as IBM used in their response whitepaper) and different database hardware is used. Also, readers need to take into account we are using just a single blade in these tests as the application server, and not four blades as in our original test. The script flow is as follows:

1. Login
2. Buy
3. Portfolio
4. Sell
5. Logout

As before, in both .NET and WebSphere test runs, we run the test for a warm-up period (to get to steady state), then capture results for a 30 minute measurement period. Unlike IBM's response paper, where they report ongoing errors returned, we see **no errors** returned for the .NET StockTrader.

In short, these tests show IBM's response benchmark results were not accurate; and verify the results of our original testing, which we continue to stand behind. To provide that evidence, we include here the Mercury LoadRunner analysis files showing a variety of statistics (including error rates, which were zero for both platforms); as well as CPU utilization charts for the application server and database during the actual WebSphere and .NET test runs.

Testing Buys and Sells

Testing sell operations via an HTTP stress test tool such as Mercury is a bit tricky, because the Holding ID for the holding to be sold must be extracted from the HTML on the portfolio page. In our test script, we used LoadRunner scripts that perform a buy operation, then visit the Portfolio page. From the portfolio page, we extract the most recent Holding ID from the returned HTML (in the script); then pass this onto

the sell page to sell that holding. It is unclear how IBM accomplished sell operations, although a similar strategy must have been employed.

In addition, since .NET StockTrader has an additional page that allows users to specify the number of shares to hold (the IBM WebSphere Trade application only allows the entire holding to be sold); we slightly modified the StockTrade.aspx page to enable trades to be placed directly with one submit operation, just as the IBM WebSphere Trade application works (so the same number of HTTP submits per buy and per sell is the same for both test runs). The .NET StockTrader design is better (and allows partial holding sells), but via the modified StockTrade.aspx page, can operate in the same way as the IBM WebSphere Trade design for fair benchmarking analysis. The other option, of course, is to remove the added functionality found in the .NET StockTrader application, and process orders with a single page ala the WebSphere Trade application. The slight modifications made to the StockTrade.aspx page to allow orders to be processed in the same way as IBM WebSphere Trade application are shown below:

```
using System;
using System.Web;
using System.Web.Security;
using System.Text;
using Trade.StockTraderWebApplicationServiceClient;
using Trade.StockTraderWebApplicationModelClasses;
using Trade.StockTraderWebApplicationSettings;
using Trade.Utility;

namespace Trade.Web
{
    /// <summary>
    /// Allows users to enter number of shares for a buy/sell operation.
    /// </summary>
    public partial class StockTrade : System.Web.UI.Page
    {
        string tradenow = null;
        protected void Page_Load(object sender, EventArgs e)
        {
            if (TextBoxID.Text == "Refresh")
                Response.Redirect(Settings.PAGE_HOME,true);
            string userid = HttpContext.Current.User.Identity.Name;
            if (userid == null)
                logout();
            Date.Text = DateTime.Now.ToString("f");
            tradenow = (string)Request["tradenow"];
            if (!IsPostBack && tradenow!="buy" && tradenow!="sell")
            {
                PanelTrade.Visible = false;
                PanelConfirm.Visible = true;
                BSLClient businessServicesClient = new BSLClient();
                string action = Request["action"];
                if (!Input.InputText(action,StockTraderUtility.EXPRESSION_NAME_10))
                    Response.Redirect(Settings.PAGE_HOME,true);
                if (action == StockTraderUtility.ORDER_TYPE_BUY)
                {
                    string quoteSymbol = Request["symbol"];
                    if (!Input.InputText(quoteSymbol,StockTraderUtility.EXPRESSION_QUOTE_ID))
                        Response.Redirect(Settings.PAGE_HOME,true);
                    QuoteDataUI quote = businessServicesClient.getQuote(quoteSymbol);
                    TradeOperation.Text = "You have requested to <b>buy</b> shares of " +
quote.quoteLink + " which is currently trading at " + quote.priceWithArrow;
                    TextBoxID.Text = quote.symbol;
                }
                else
                {
                    string holdingID = Request["holdingid"];
                    if (!Input.InputText(holdingID,StockTraderUtility.EXPRESSION_HOLDINGID))
```

```
                        Response.Redirect(Settings.PAGE_HOME,true);
                    int holdingid = Convert.ToInt32(holdingID);
                    if (action == StockTraderUtility.ORDER_TYPE_SELL)
                    {
                        if (Settings.interfaceMode ==
StockTraderUtility.ACCESS_WebService_WebSphere || Settings.interfaceMode ==
StockTraderUtility.ACCESS_WebService_WLS || Settings.interfaceMode ==
StockTraderUtility.ACCESS_WebService_OC4J)
                        {
                            TradeOperation.Text = "You have requested to sell your holding " +
holdingID + ". Please confirm this request.";
                            //indicate for postback we are running against WebSphere Trade 6.1
which does not implement the functionality/business logic
                            //to sell a portion of a holding--only the entire holding can be sold
at once.
                            quantity.Text = "-1";
                            quantity.Visible = false;
                            ButtonTrade.Text = "  Sell  ";
                            TextBoxID.Text = holdingID;
                        }
                        else
                        {
                            HoldingDataUI holding = businessServicesClient.getHolding(userid,
holdingid);
                            StringBuilder strBldr = new StringBuilder("You have requested to sell
all or part of your holding ");
                            strBldr.Append(holdingID);
                            strBldr.Append(". This holding has a total of ");
                            strBldr.Append(holding.quantity);
                            strBldr.Append(" shares of stock <b>");
                            strBldr.Append(holding.quoteID);
                            strBldr.Append("</b>. Please indicate how many shares to sell.");
                            TradeOperation.Text = strBldr.ToString();
                            quantity.Text = holding.quantity;
                            ButtonTrade.Text = "  Sell  ";
                            TextBoxID.Text = holdingID;
                        }
                    }
                }

            }
            else
            {
                PanelTrade.Visible = true;
                PanelConfirm.Visible = false;
                if (tradenow == "buy" || tradenow == "sell")
                    ButtonTrade_Click(null, null);

            }
        }

        private void logout()
        {
            FormsAuthentication.SignOut();
            Response.Redirect(Settings.PAGE_LOGIN, true);
        }

        protected void ButtonTrade_Click(object sender, EventArgs e)
        {
            string symbol = null;
            int holdingID = -1;
            if (tradenow == "sell")
            {
                ButtonTrade.Text = "Sell";
                holdingID = Convert.ToInt32((string)Request["holdingid"]);
            }
            else if (tradenow == "buy")
            {
                ButtonTrade.Text = "Buy";
                symbol = Request["symbol"];
            }
```

```
            string userid = HttpContext.Current.User.Identity.Name;
            if (userid == null)
                logout();
            Date.Text = DateTime.Now.ToString("f");
            BSLClient businessServicesClient = new BSLClient();
            OrderDataUI order = null;
            string action = Request["action"];
            if (!Input.InputText(action, StockTraderUtility.EXPRESSION NAME 10))
                Response.Redirect(Settings.PAGE_HOME, true);
            double quantityTrade = 0;


            quantityTrade = Convert.ToDouble(quantity.Text);
            if (ButtonTrade.Text.Contains("Buy"))
            {
                if (symbol==null)
                    symbol = TextBoxID.Text;
                order = businessServicesClient.buy(userid, symbol, quantityTrade);
            }
            else if (ButtonTrade.Text.Contains("Sell"))
            {
                if (holdingID==-1)
                    holdingID = Convert.ToInt32(TextBoxID.Text);

                //WebSphere Trade 6.1/Oracle Trade do not provide functionality for trading
partial holdings.
                //.NET StockTrader does, but will default to Trade 6.1 behavior and sell and
entire holding if no
                //quantity parameter is detected on the query string.  Here we check if a
quantity parameter actually
                //exists on the query string.  This is a bit of extra overhead, but done so we
can add functionality
                // (selling part of a holding) yet not sacrifice interop with WebSphere Trade
6.1/Oracle WebLogic and
                //OC4J implementations.
                if (quantityTrade==-1)
                    quantityTrade = 0;  //Value of 0 indicates to sell entire holding.
                order = businessServicesClient.sell(userid, holdingID, quantityTrade);
            }
            else
                //Goodbye! Only valid ops are buy and sell. This is a harsh
                //penalty for trying to be tricky.
                Response.Redirect(Settings.PAGE_LOGOUT);
            if (order != null)
                {
                    Cache.Remove(Settings.CACHE_KEY_CLOSED_ORDERSALERT + userid);
                    string orderIdStr = order.orderID.ToString();
                    OrderID.Text = orderIdStr;
                    OrderStatus.Text = order.orderStatus;
                    OpenDate.Text = order.openDate.ToString();
                    CompletionDate.Text = order.completionDate.ToString();
                    OrderFee.Text = string.Format("{0:C}", order.orderFee);
                    OrderType.Text = order.orderType;
                    string orderLink = order.quoteLink;
                    Symbol.Text = orderLink;
                    string orderQty = string.Format("{0:0,0}", order.quantity);
                    QtyTraded.Text = orderQty;
                    StringBuilder strBuilder = new StringBuilder("Order <b>");
                    strBuilder.Append(orderIdStr);
                    strBuilder.Append("</b> to ");
                    strBuilder.Append(order.orderType);
                    strBuilder.Append(" ");
                    strBuilder.Append(orderQty);
                    strBuilder.Append(" shares of ");
                    strBuilder.Append(orderLink);
                    strBuilder.Append(" has been submitted for processing.<br/><br/>");
                    strBuilder.Append("Order Details:");
                    ConfirmMessage.Text = (strBuilder.ToString());
                }
                else
                    ConfirmMessage.Text = StockTraderUtility.EXCEPTION_MESSAGE_BAD_ORDER_RETURN;
```

```
            TextBoxID.Text = "Refresh";
            }


        protected void ButtonCancel_Click(object sender, EventArgs e)
        {
            Response.Redirect(Settings.PAGE_HOME, true);
        }
}
}
```

The scripts used for buy and sell operations are shown below, for both .NET and WebSphere:

.NET Buy LoadRunner Script

```
#include "as_web.h"
#include "lrw_custom_body.h"

Buy()

{

lr_think_time(1);

lr_start_transaction("Buy");

web_submit_data("StockTrade.aspx",

                        "Action=http://{Server}/Trade/StockTrade.aspx?action=buy&tradenow=buy&symbol=s:{Symbol}",

                        "Method=POST",

                        "RecContentType=text/html",

                        "Referer=http://192.168.4.36/Trade/StockTrade.aspx?action=buy&symbol=s:1",

                        "Snapshot=t7.inf",

                        "Mode=HTTP",

                        ITEMDATA,

                        "Name=quantity", "Value=110", ENDITEM,

                        "Name=ButtonTrade", "Value=  Buy  ", ENDITEM,

                        "Name=TextBoxID", "Value=s:0", ENDITEM,

                        LAST);

                        lr_end_transaction("Buy", LR_AUTO);

return 0;

}
```

WebSphere 7 Buy LoadRunner Script

```
#include "as_web.h"
#include "lrw_custom_body.h"

Buy()

{

lr_think_time(1);

lr_start_transaction("Buy");

web_submit_data("app3",

                "Action=http://{Server}:{port}/Trade7/app",

                "Method=POST",

                "RecContentType=text/html",

                "Referer=http://{Server}:{port}/Trade7/app?action=quotes&symbols=s:0",

                "Snapshot=t36.inf",

                "Mode=HTTP",

                ITEMDATA,

                "Name=action", "Value=buy", ENDITEM,

                "Name=symbol", "Value=s:{Symbol}", ENDITEM,

                "Name=quantity", "Value=100", ENDITEM,

                LAST);

lr_end_transaction("Buy", LR_AUTO);

return 0;

}
```

.NET Sell LoadRunner Script

```
Sell()

{

lr_think_time(1);

lr_start_transaction("Portfolio");

web_set_max_html_param_len("4690");

        web_reg_save_param("WCSParam1",

                "LB/IC=StockTrade.aspx?action=sell&holdingid=",

                "RB/IC=\">Sell",
```

```
                        "Ord=1",
                        "Search=body",
                        LAST);
        web_url("Sell1",
                        "URL=http://{Server}/TRADE/portfolio.aspx",
                        "Resource=0",
                        "RecContentType=text/html",
                        "Referer=http://{Server}/TRADE/Order.aspx",
                        "Snapshot=t82.inf",
                        "Mode=HTML",
                        LAST);
lr_end_transaction("Portfolio", LR_AUTO);
lr_think_time(1);
lr_start_transaction("Sell");
        web_url("Sell2",

        "URL=http://{Server}:{port}/TRADE/StockTrade.aspx?tradenow=sell&action=sell&quantity=100.00&holdingID={WCSParam1}&symbol=s:98",
                        "Resource=0",
                        "RecContentType=text/html",
                        "Referer=http://192.168.4.142/TRADE/app?action=portfolio",
                        "Snapshot=t83.inf",
                        "Mode=HTTP",
                        LAST);


lr_end_transaction("Sell", LR_AUTO);
return 0;
}
```

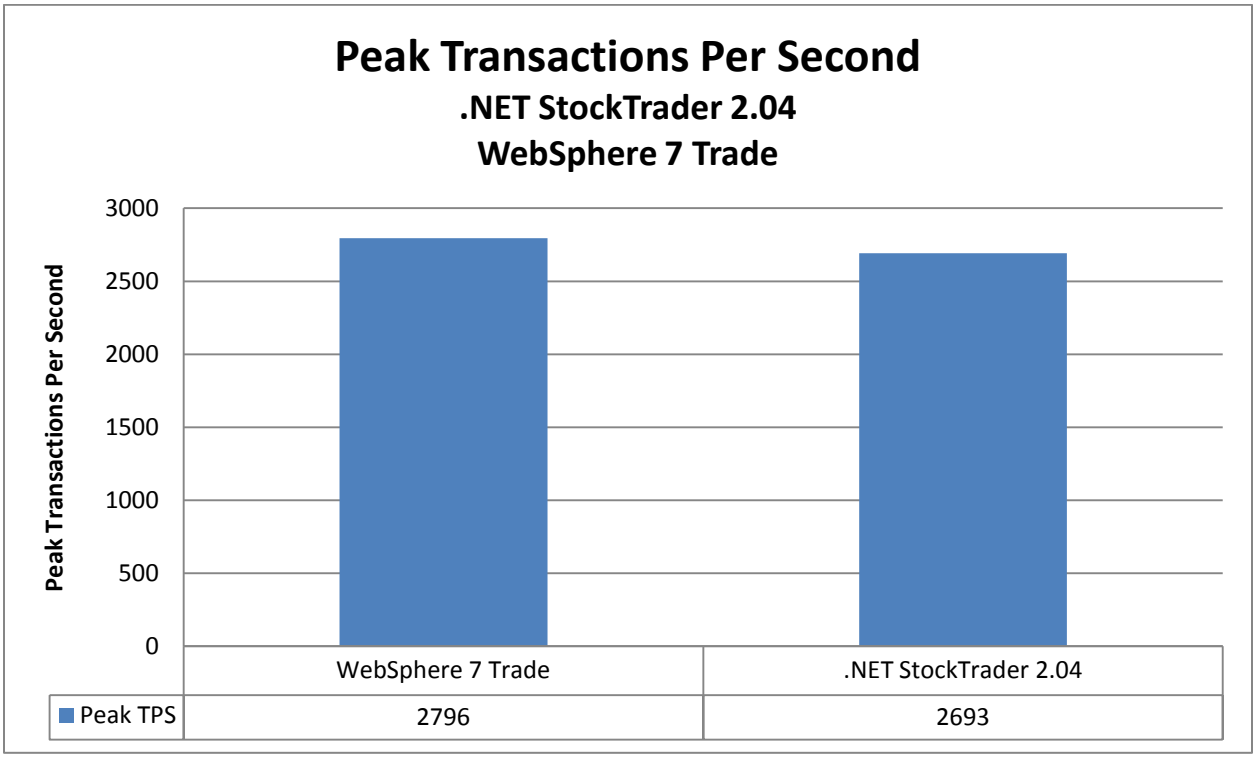
WebSphere 7 Sell LoadRunner Script

```
Sell()
{
```

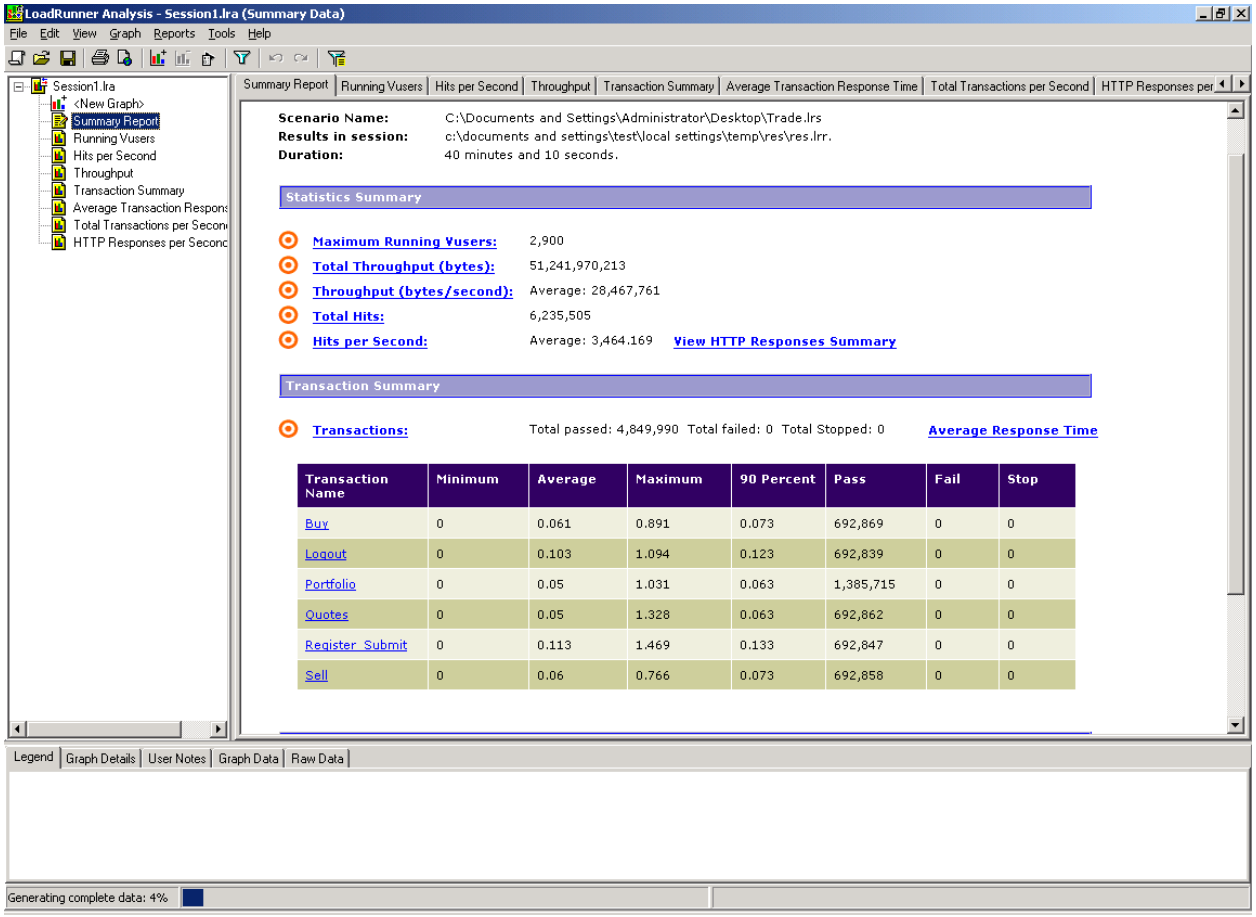
```
lr_think_time(1);

lr_start_transaction("Portfolio");

web_set_max_html_param_len("4690");

	web_reg_save_param("WCSParam1",

		"LB/IC=<A href=\"app?action=sell&holdingID=",

		"RB/IC=\"",

		"Ord=1",

		"Search=body",

		"RelFrameId=1",

		LAST);



web_url("Sell1",

		"URL=http://{Server}:{port}/Trade7/app?action=portfolio",

		"TargetFrame=",

		"Resource=0",

		"RecContentType=text/html",

		"Referer=http://192.168.4.123/Trade7/app?action=account",

		"Snapshot=t38.inf",

		"Mode=HTML",

		LAST);



lr_end_transaction("Portfolio", LR_AUTO);

lr_think_time(1);

	lr_start_transaction("Sell");

	web_url("Sell2",

		"URL=http://{Server}:{port}/Trade7/app?action=sell&holdingID={WCSParam1}",

		"Resource=0",

		"RecContentType=text/html",

		"Referer=http://192.168.4.142/Trade7/app?action=portfolio",

		"Snapshot=t77.inf",

		"Mode=HTTP",
```

```
                        LAST);



lr_end_transaction("Sell", LR_AUTO);

return 0;

}
```

Benchmark Results



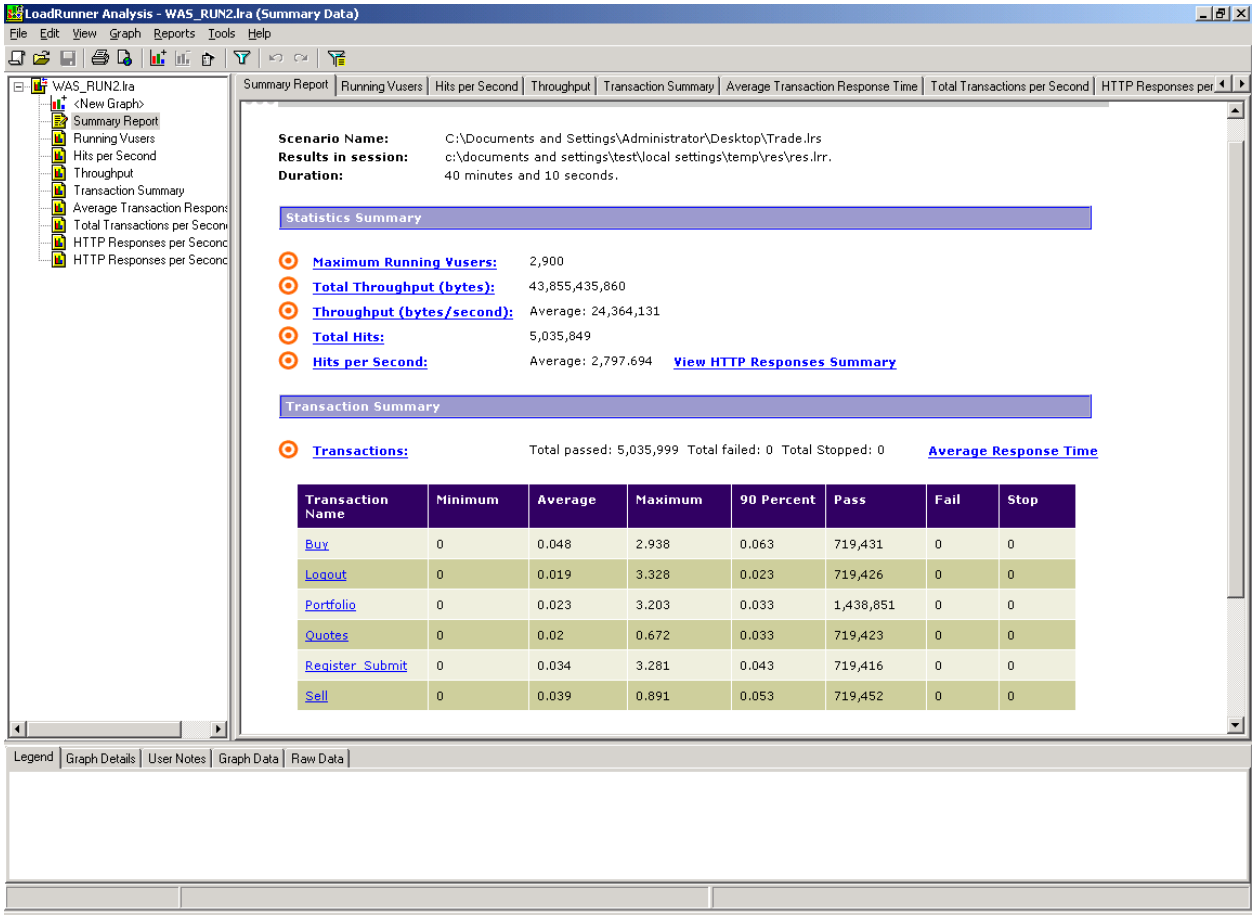
.NET StockTrader LoadRunner Summary



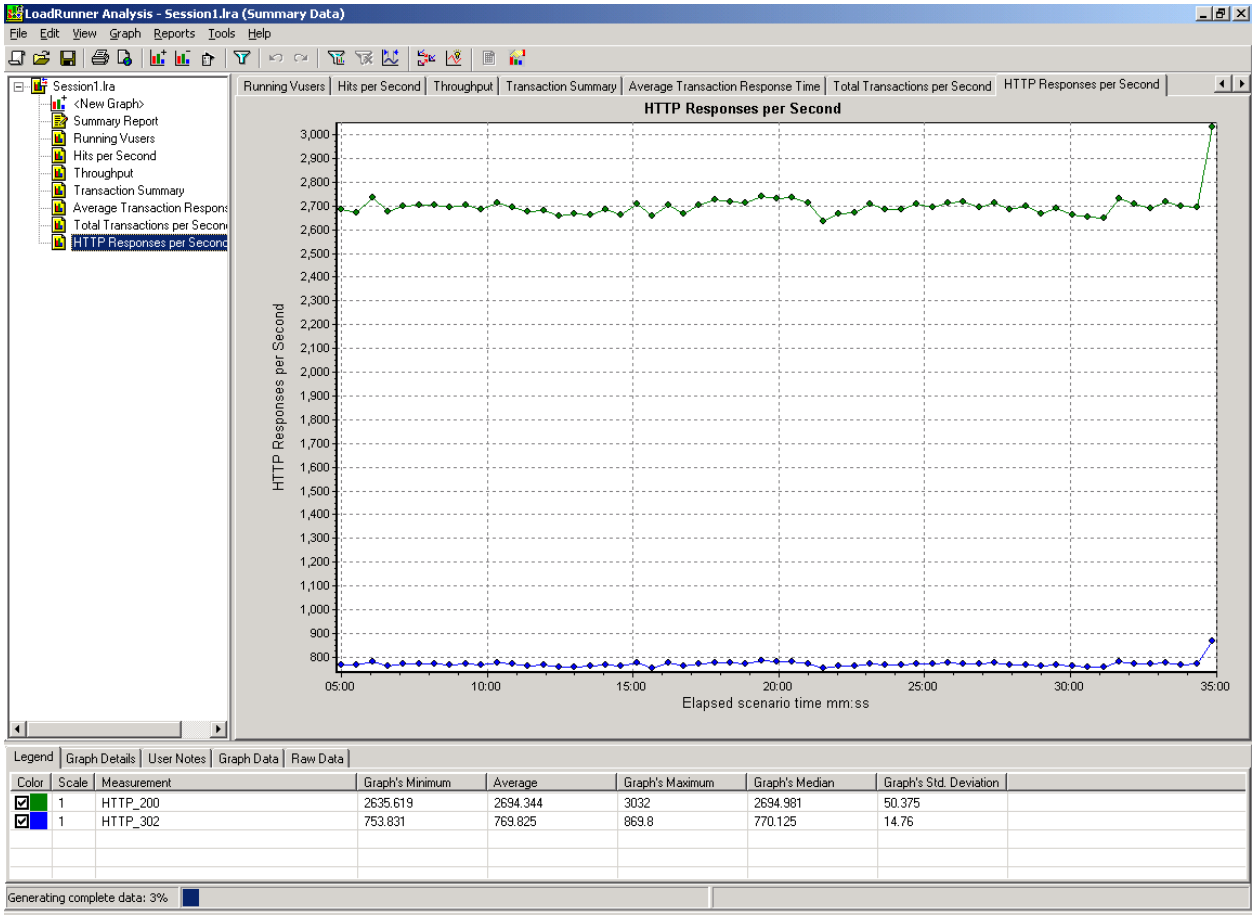
Note: .NET StockTrader performs a server Response.Redirect in the ASPX code-behind on both the login page (processing the login, then on successful login directing to the TradeHome.aspx page after proper encrypted ASP.NET forms authentication ticket information is set; and on logout, redirecting to the login page after closing out the user session). This is a standard design for ASP.NET. Because LoadRunner captures the login and logout pages each **as two separate http responses** (one for the login page with a status of HTTP 302-redirect; one for the Trade home page as HTTP 200, for example); LoadRunner records two **hits per second** for each of these pages.

The IBM WebSphere Trade 7 application uses Servlets to process the login, and the Servlet response on successful login is simply one HTTP 200 code. So LoadRunner sees each of these pages as a single page hit each, even though precisely equivalent user interactions are taking place as with the .NET StockTrader script. Hence, LoadRunner records hits per second as higher for .NET StockTrader than IBM WebSphere 7 Trade; even though the functionality is the same. In all measured results (both in this response and our original paper); we capture **transactions per second**, not hits per second; which is the real measure of the two applications performing the same user interaction with the application throughout the test runs. This is the metric that should be compared (and was), or else the .NET StockTrader results would be artificially inflated.

WebSphere 7 Trade LoadRunner Summary



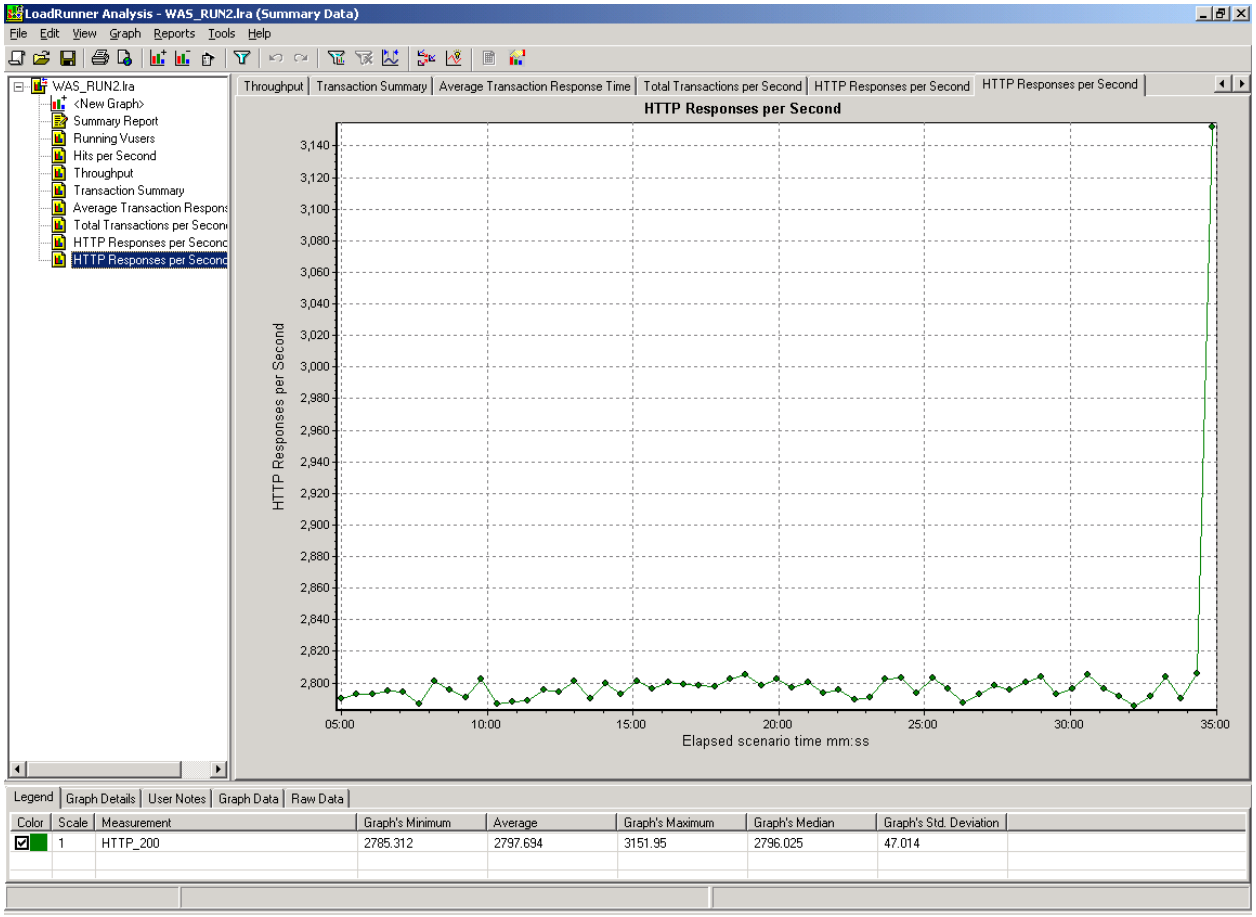
.NET StockTrader HTTP Responses/Second



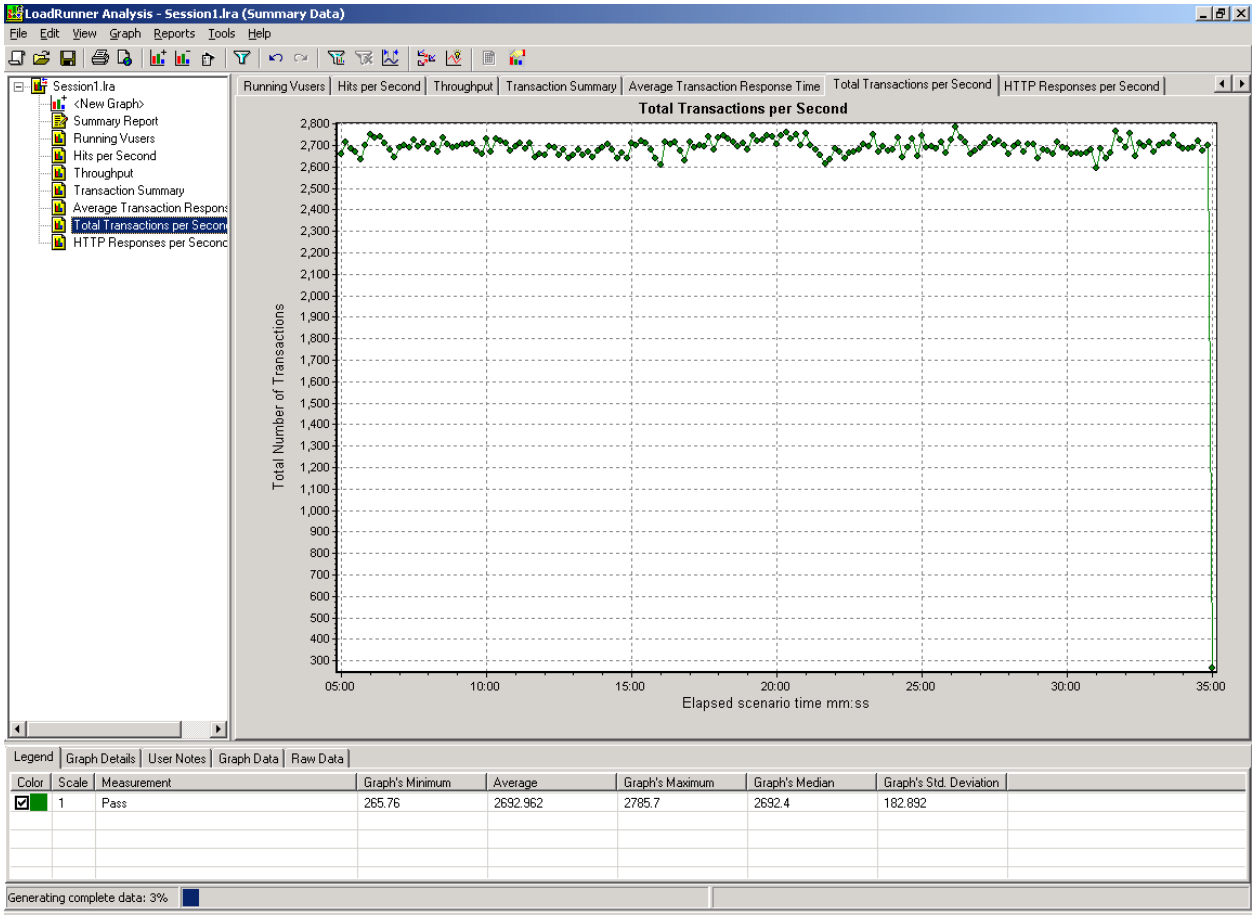
Note: Here you see the HTTP 200 status codes exactly equating to transactions per second, per the previous note about the impact of Response.Redirect on login and logout pages for the .NET StockTrader.

Also note there are zero errors returned during the entire test run, showing the .NET StockTrader does not have errors under load, unlike the IBM claims in their response document.

IBM WebSphere 7 HTTP Responses/Second

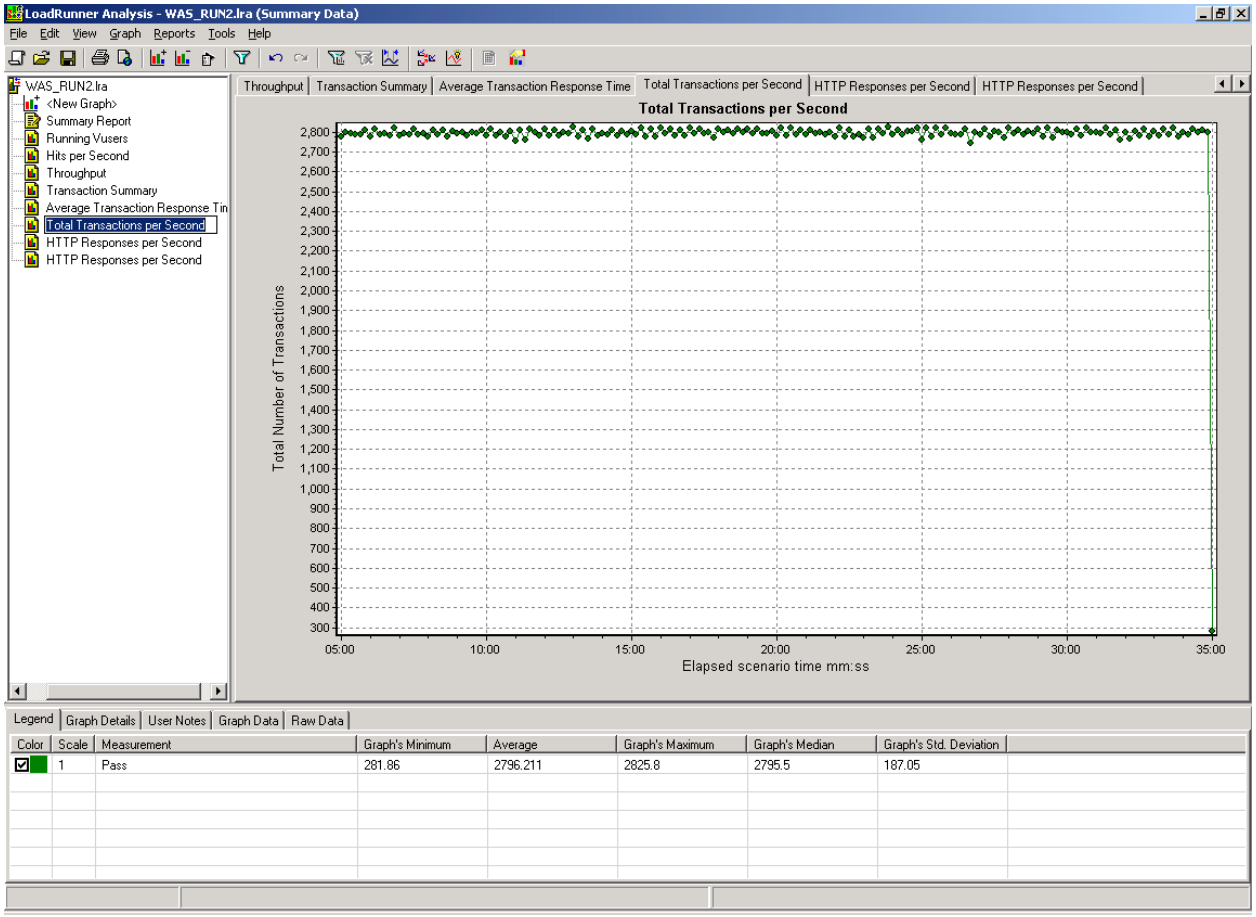


.NET StockTrader Pass/Fail Transactions per Second



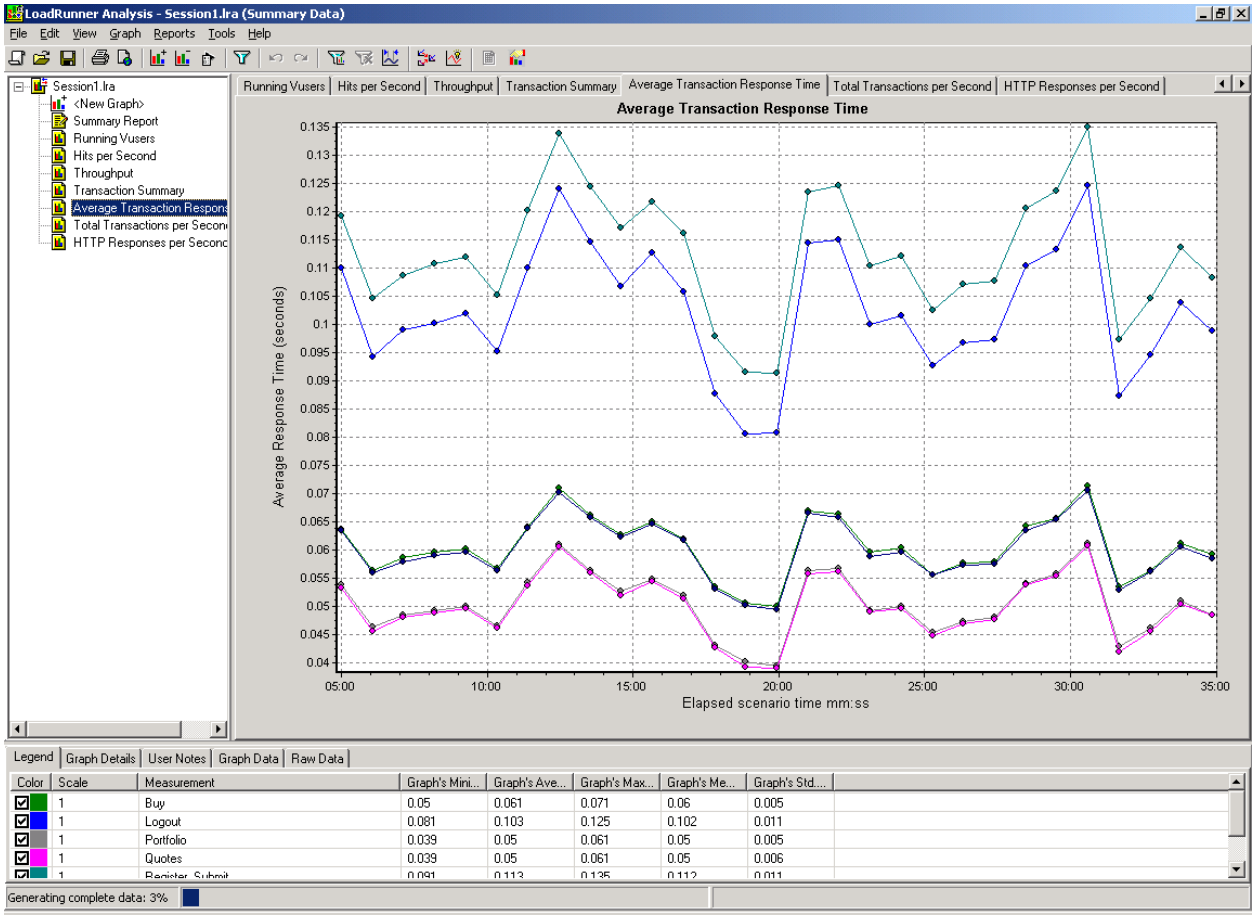
Note: There are zero failed transactions.

IBM WebSphere 7 Trade Pass/Fail Transactions per Second



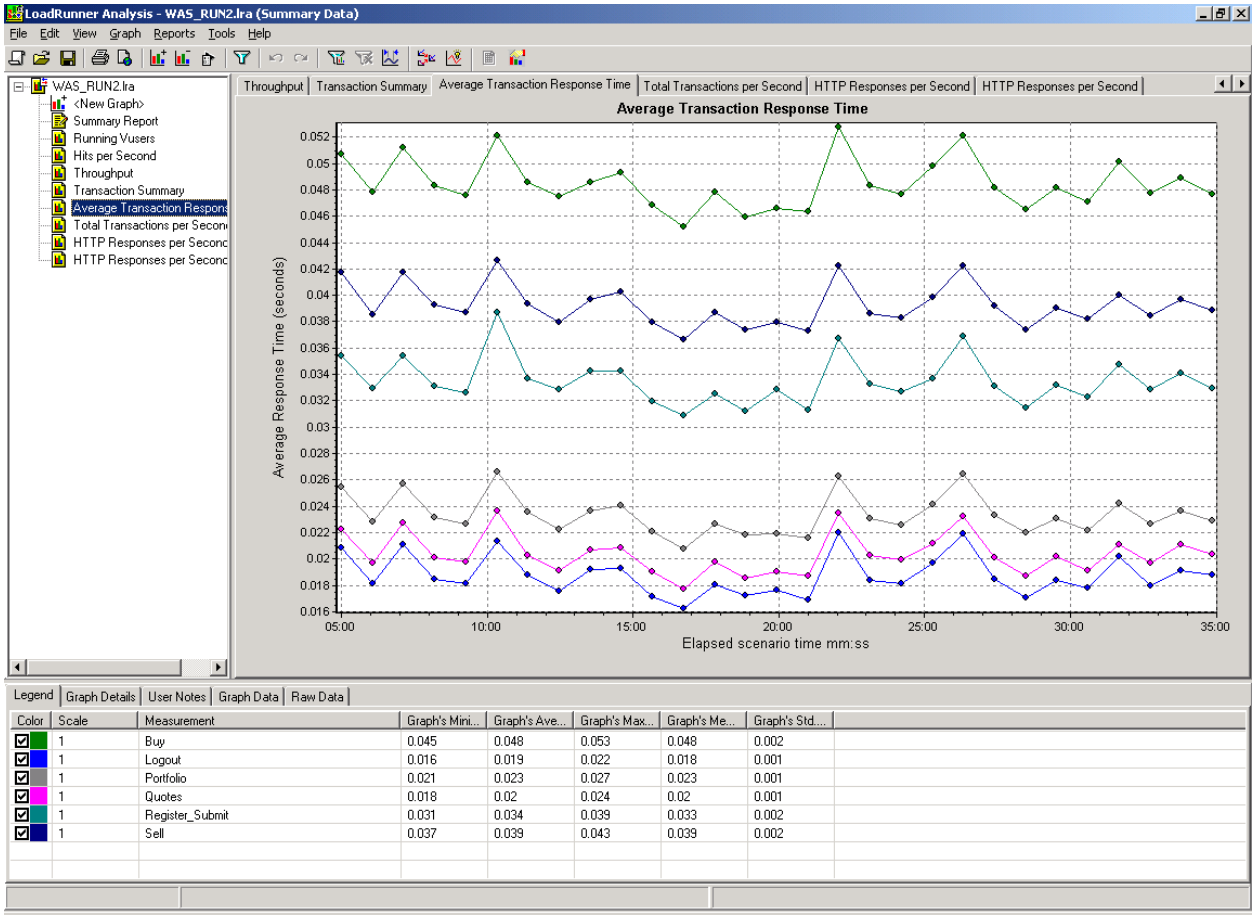
Note: There are zero failed transactions, just like the .NET StockTrader test.

.NET StockTrader Transaction Response Times



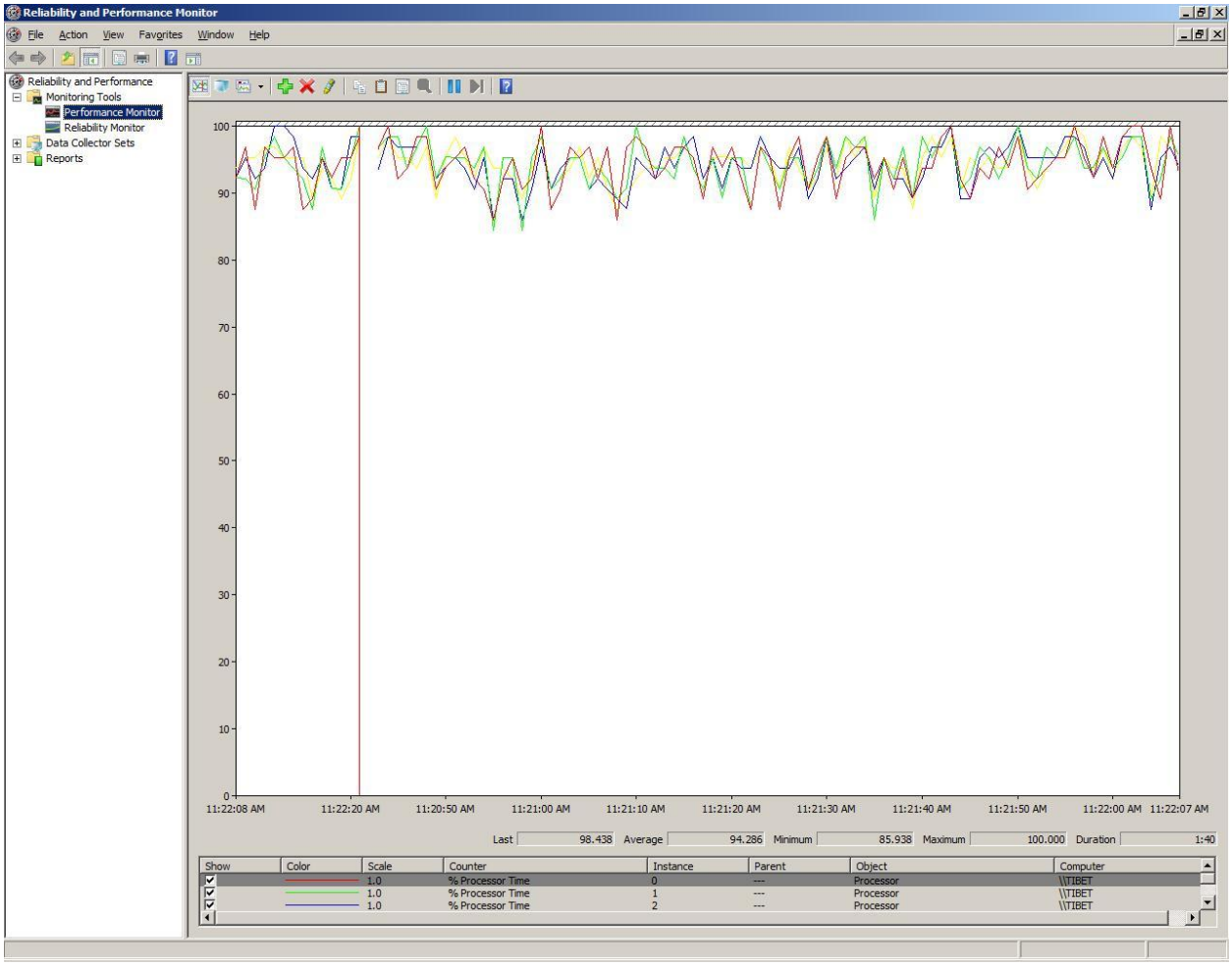
Note: Response times are consistently flat (less than .005 seconds) until server saturation is hit, then as is typical, queuing starts and response times become a factor of the queue size. In the image above, we show response times for the 30 minute measurement interval at a consistent user load of 2900 simulates users (1 second tx think time between requests). At 2900, request queuing has just started to kick in as the server is at full saturation for the test script.

IBM WebSphere Trade 7 Transaction Response Times

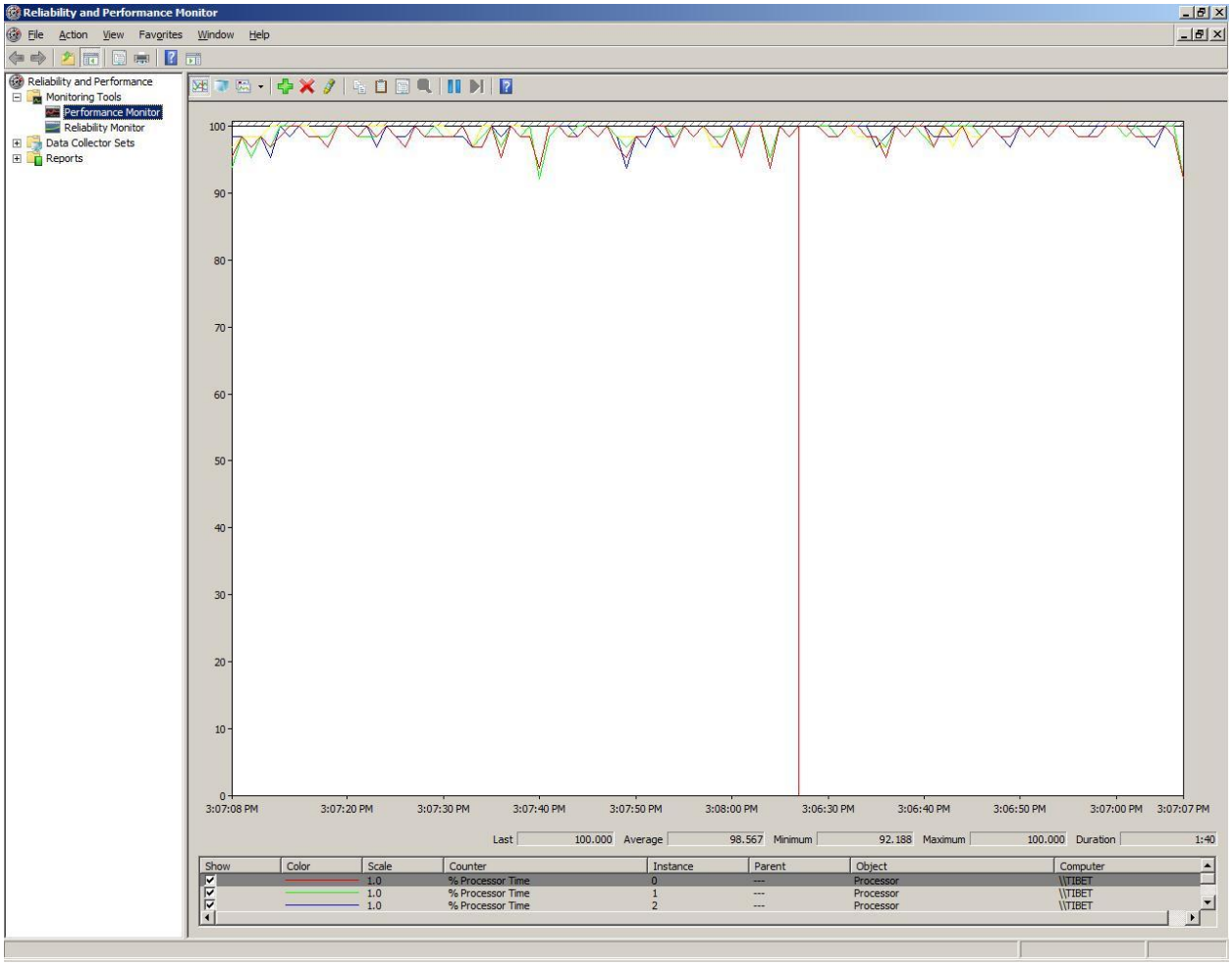


Note: Response times are consistently flat (less than .005 seconds) until server saturation is hit, then as is typical, queuing starts and response times become a factor of the queue size. In the image above, we show response times for the 30 minute measurement interval at a consistent user load of 2900 simulates users (1 second tx think time between requests). At 2900, request queuing has just started to kick in as the server is at full saturation for the test script.

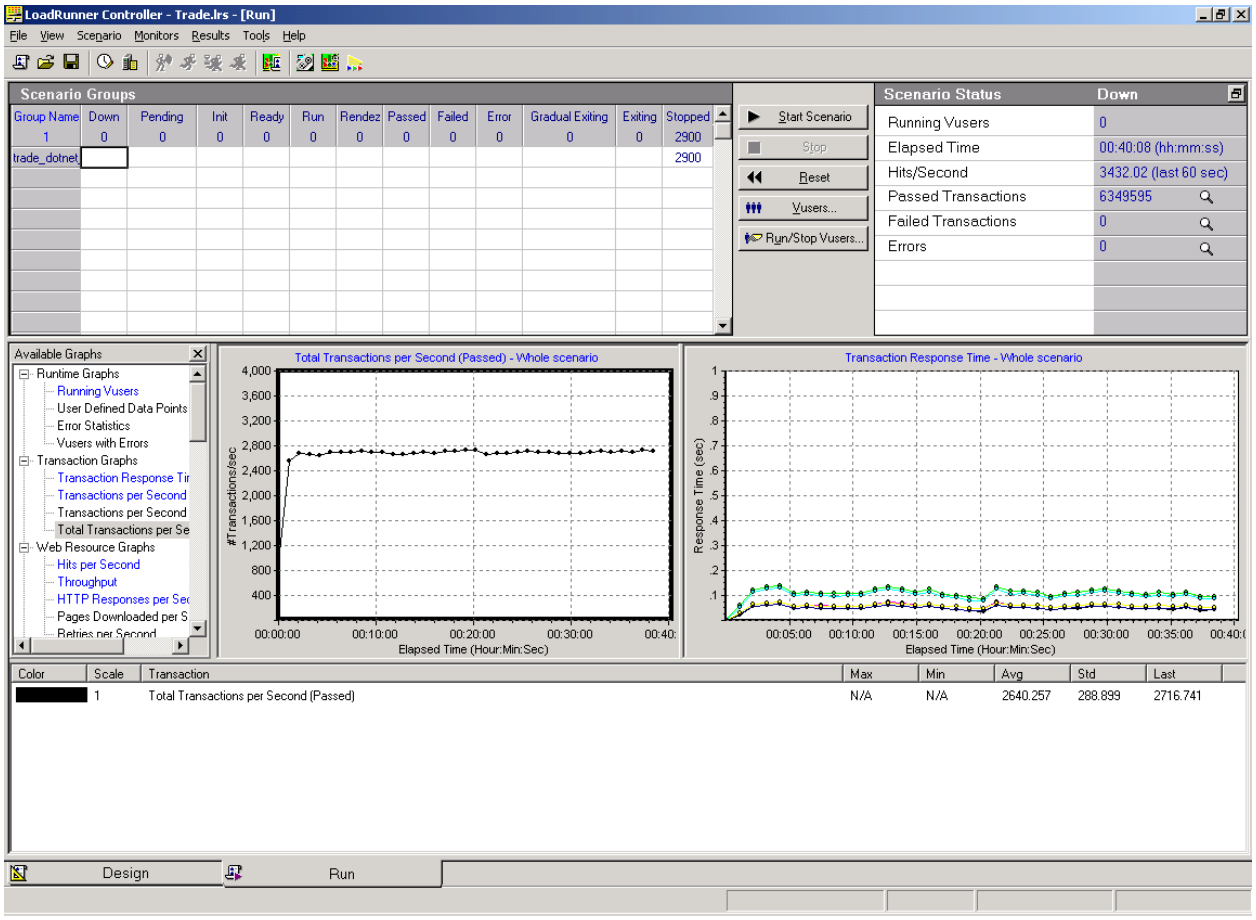
.NET StockTrader Application Server CPU Utilization during Test Run



IBM WebSphere 7 Trade Application Server CPU Utilization during Test Run

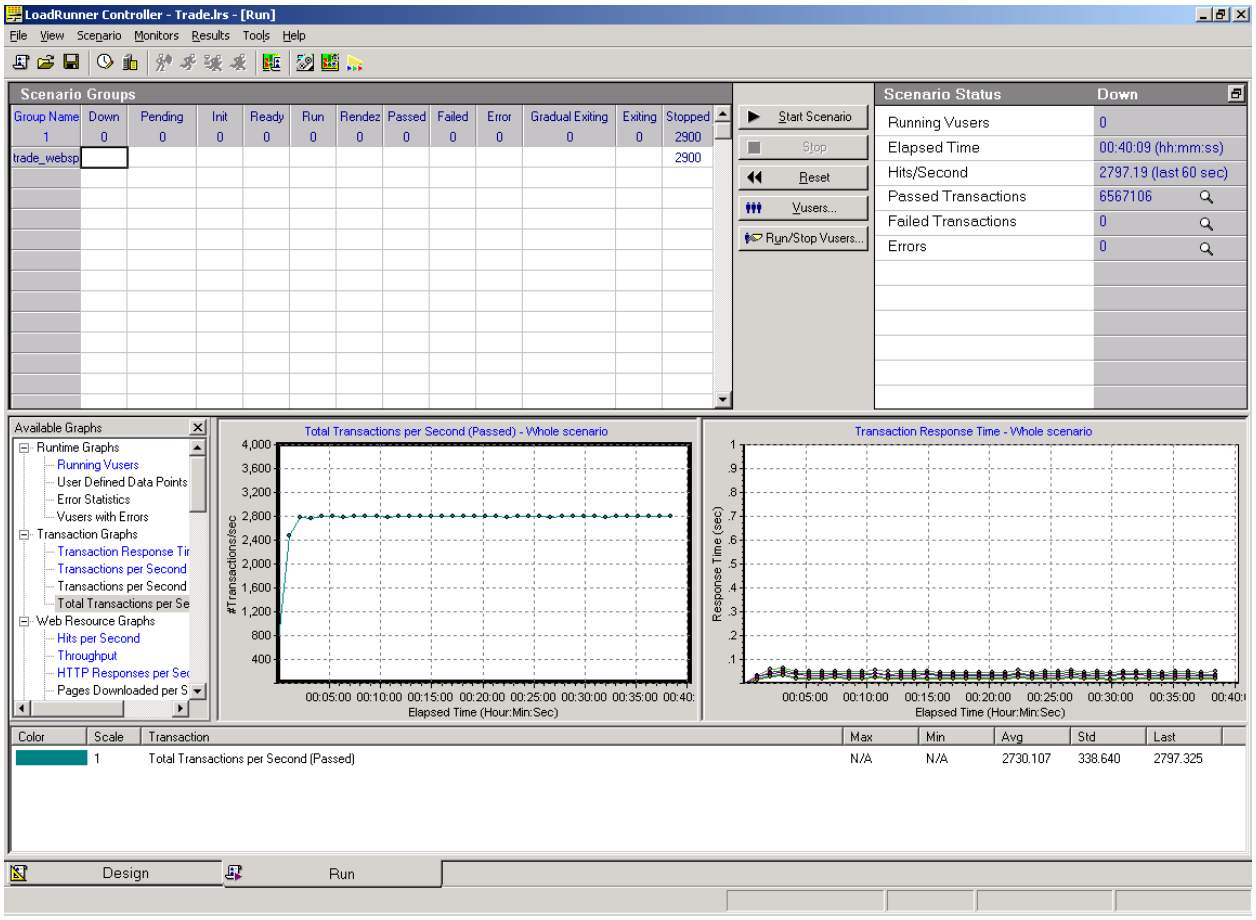


.NET StockTrader Application Server LoadRunner Summary on Completion of Test



Note: This is inclusive of short warm-up period, and thus includes all results before filtering to just the 30 minute steady-state measurement interval.

IBM WebSphere 7 Trade Application LoadRunner Summary on Completion of Test



Note: This is inclusive of short warm-up period, and thus includes all results before filtering to just the 30 minute steady-state measurement interval.

Appendix B: Web Service Tests using no HTTP Server

As noted earlier, one of the IBM claims in their benchmark response document is that in the Web Service workloads (StockTrader and WSTest), we inserted an unnecessary HTTP server into the system under test. The use of this HTTP Server was fully discussed in the original benchmark paper, and is done in accordance with IBM's own best practice deployment guidelines for WebSphere. In such a setup, IBM recommends using the IBM HTTP Server (Apache) as the front end Web Server, which then routes requests to the IBM WebSphere Application server. In our tests, we co-located this HTTP on the same machine as the Application Server. This is equivalent to the .NET/WCF Web Service tests, where we hosted the WCF Web Services in IIS 7, with co-located IIS 7 HTTP Server routing requests to the .NET application pool processing the WCF service operations. So in both tests, we tested an equivalent setup, using IBM HTTP Server (Apache) as the front end to WebSphere/JAX-WS services; and Microsoft IIS 7 as the front end to the .NET/WCF services. Therefore, we stand behind all our original results.

However, it is true that in the Web Service tests, the benchmark client (the Capacity Planner tool as included with the .NET StockTrader 2.04 download) could have been configured to bypass the IBM HTTP Server, and instead route requests directly to the IBM WebSphere Application Server in-process HTTP Listener (port 9080); and this will result in higher requests per second for IBM WebSphere. However, in such a setup the proper equivalent configuration for .NET is to self-host the WCF services, and also bypass the IIS 7 front-end Web Server. In the interest of completeness, and to respond to IBM's claim, we include results here for WSTest running:

1. With IBM's HTTP Server (Apache) as the front end to WebSphere JAX-WS services
2. With IIS 7 as the front end to .NET/WCF services
3. Without IBM HTTP Server; benchmark clients talking directly to IBM WebSphere over in-process port 9080
4. Without IIS 7; benchmark clients talking directly to the in-process WCF self-hosted port.

Benchmark runs 1 and 2 can be compared as one equivalent configuration; and benchmark runs 3 and 4 can be compared as a second equivalent configuration. The results documented below show both platforms produce higher TPS rates (Web Service Requests per Second) without a front end Web Server; .NET/WCF significantly outperforms IBM WebSphere in the HTTP Server (recommended) configuration; and without an HTTP Server performance is about equivalent, with WebSphere very slightly outperforming self-hosted .NET/WCF services.

The application server test hardware is the same as used in the previous StockTrader/Trade 7 results.

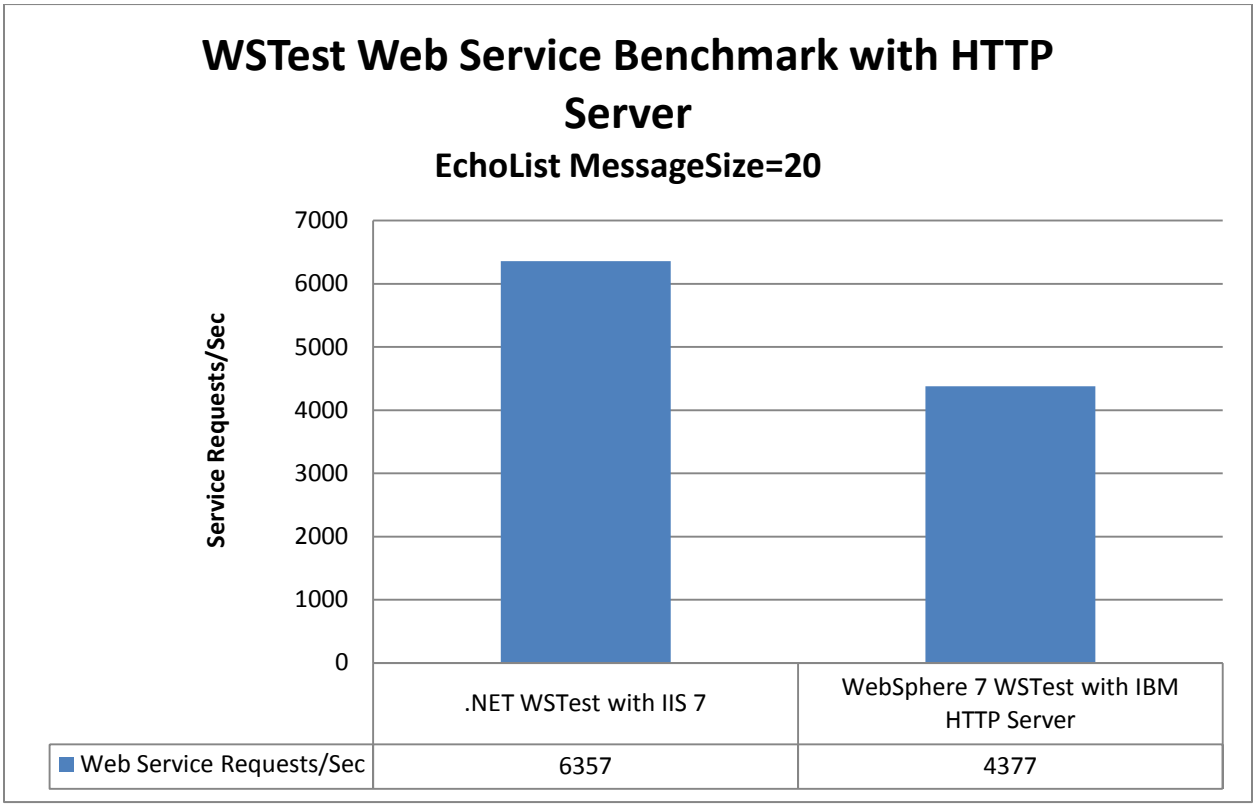
Application Server Hardware

1 HP ProLiant BL460c
1 Quad-core Intel Xeon E5450 CPU (3.00 GHz)
32 GB RAM
2 x 1GB NICs
Windows Server 2008 64-bit

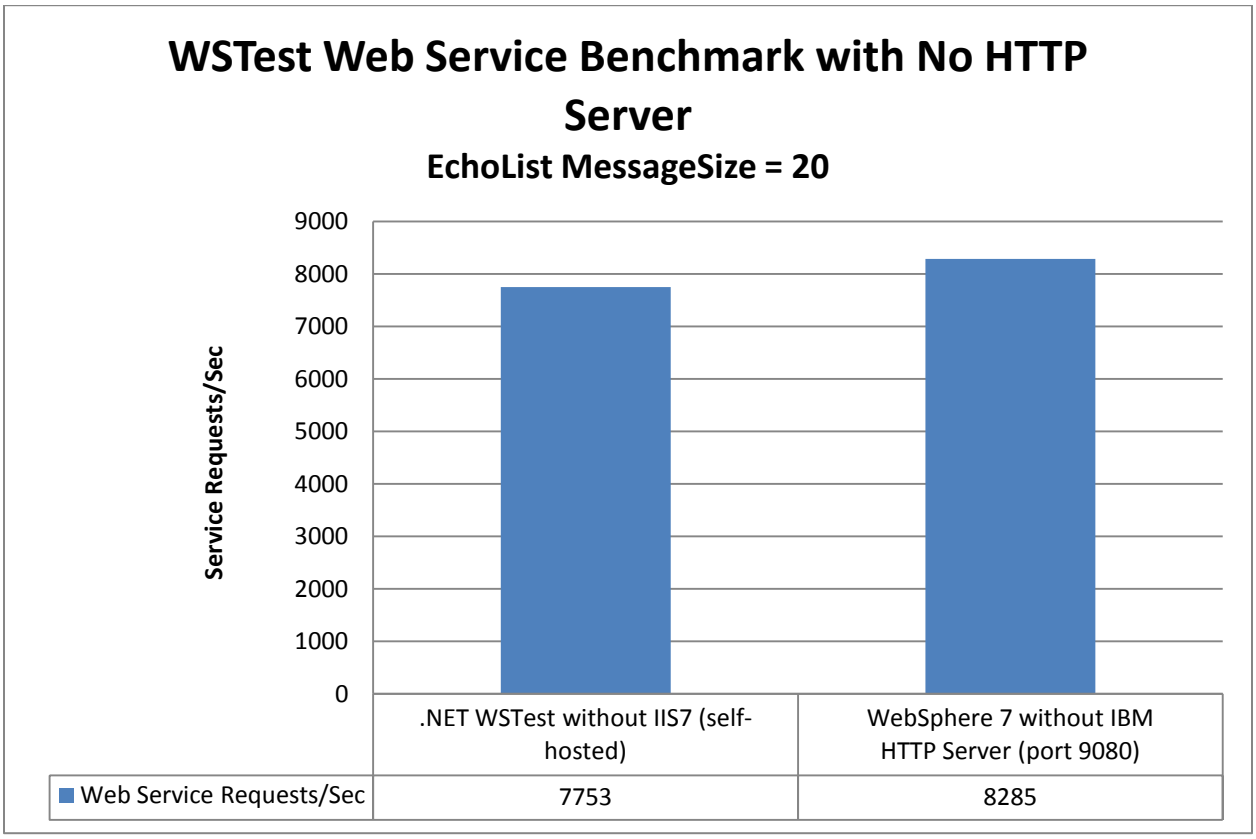
.NET 3.5 (SP1) 64-bit
IBM WebSphere 64-bit

The self-hosted implementations for .NET StockTrader and .NET WSTest implementations are included with the original .NET StockTrader download, as is the IBM WebSphere Java source code for the JAX-WS services for these two workloads. So customers already have everything they need to replicate/verify the results shown below.

.NET WSTest EchoList Front Ended with IIS 7 vs. IBM WebSphere 7 WSTest EchoList Front Ended with IBM HTTP Server



.NET WSTest EchoList without IIS 7 (self-hosted WCF HTTP Service) vs. IBM WebSphere 7 WSTest EchoList without IBM HTTP Server (in process WebSphere port 9080)



WebSphere Tuning – Windows Server 2008

Servlet Caching turned on in Web Container

Session State set to 5 minute expiration (in-process session state)

Access Log Turned Off

Performance Monitor Infrastructure Turned Off

App Profile Service Off

Diagnostic Trace Turned Off

System Out Off

JVM Configured not to write System.Out printline messages

HTTP Channel maximum persistent requests = -1

Minimum Web Container threads = 50

Maximum Web Container threads = 50

Minimum ORB threads = 24

Maximum ORB threads = 24

Minimum Default threads = 20

Maximum Default threads = 20

Minimum JDBC Connections in Pool = 51

Maximum JDBC Connections in Pool = 51

SQL Statement Cache Size = 150

Java Heap Size: 2000 min/2100 MB max

- **Xgcthreads8** as JVM startup parameter (note, this setting had no effect; nor did –Xgcthreads4; but we used to be consistent with IBM's testing)

IBM HTTP Server Tuning Windows Server 2008

Access Log Off

Max KeepAlive Requests unlimited

Default Threads 3200

Threads/Child 3200

.NET Tuning

.NET Worker Process

Rapid Fail Protection off
Pinging off
Recycle Worker Process off

Maximum Queue Size = 1000

ASP.NET

Forms Authentication with protection level=”All”

Forms Authentication Timeout=5 minutes

IIS 7.0 Virtual Directory

Authentication Basic Only
Access Logging Off

Service Behavior for Trade Business Services:

<behavior name="TradeServiceBehaviors">
<serviceDebug httpHelpPageEnabled="true" includeExceptionDetailInFaults="true"/>
<serviceMetadata httpGetEnabled="true" httpGetUrl=""/>
<serviceThrottling maxConcurrentInstances="400" maxConcurrentCalls="400"/>
</behavior>

.NET StockTrader

Max DB Connections = 60

Min DB Connections = 60

DB2

Logging files expanded to 15 GB

Logging Set to One Drive Array (array a)

Database file on Second Drive Array (array b)

Max Application Connections = 400

SQL/Server

Logging files expanded to 15 GB

Logging Set to One Drive Array (array a)

Database file on Second Drive Array (array b)

This document supports the release of Windows Server® 2008 and the Microsoft .NET Framework 3.5.

The information contained in this document represents the current view of Microsoft Corp. on the issues disclosed as of the date of publication. Because Microsoft must respond to changing market conditions, this document should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented. This document is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, IN THIS DOCUMENT.

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the companies, organizations, products, domain names, e-mail addresses, logos, people, places and events depicted in examples herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Microsoft grants you the right to reproduce this guide, in whole or in part.

Microsoft may have patents, patent applications, trademarks, copyrights or other intellectual property rights covering subject matter in this document, except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights or other intellectual property.

Microsoft, Windows Server, the Windows logo, Windows, Active Directory, Windows Vista, Visual Studio, Internet Explorer, Windows Server System, Windows NT, Windows Mobile, Windows Media, Win32, WinFX, Windows PowerShell, Hyper-V, and MSDN are trademarks of the Microsoft group of companies.

The names of actual companies and products mentioned herein may be the trademarks of their respective owners.