.NET 3.0/Windows Communication Foundation and IBM WebSphere 6.1 Service-Oriented Performance and Scalability Benchmark

.NET StockTrader vs. IBM WebSphere Trade 6.1 Benchmark Results for Transactions, Web Services, and Messaging Workloads

6/4/2007

© Microsoft Corporation 2007

Microsoft .netFramework

The information contained in this document represents the current view of Microsoft Corporation on the issues discussed as of the date of publication. Because Microsoft must respond to changing market conditions, it should not be interpreted to be a commitment on the part of Microsoft, and Microsoft cannot guarantee the accuracy of any information presented after the date of publication.

This White Paper is for informational purposes only. MICROSOFT MAKES NO WARRANTIES, EXPRESS OR IMPLIED, AS TO THE INFORMATION IN THIS DOCUMENT.

Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

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.

© 2007 Microsoft Corporation. All rights reserved.

Microsoft, the .NET logo, Visual Studio, Win32, Windows, and Windows Server 2003 are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

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

Microsoft Corporation • One Microsoft Way • Redmond, WA 98052-6399 • USA

Contents

Introduction

This paper presents detailed benchmark results based on extensive performance and scalability testing of IBM WebSphere 6.1.07 Network Deployment Edition and .NET with the Windows Communication Foundation (WCF). The benchmark focuses on three core workloads:

- Web Services
- Message-oriented transaction processing
- Data-driven Web application with middle-tier transaction services and data access

The benchmark tests focus on comparing an end-to-end solution based on the IBM Trade 6.1 performance application. Trade 6.1 is a J2EE-based application developed by IBM as a best-practice performance sample application and capacity testing tool for IBM WebSphere 6.1. The application is available for free download from the IBM WebSphere performance site, and is used extensively by IBM throughout most of their core enterprise Redbooks for WebSphere. Since the application was designed specifically as a performance-driven application, it presents a good opportunity to compare the performance of IBM WebSphere to the performance of .NET/Windows Server 2003 running an application server workload.

.NET StockTrader Sample Application and Performance Kit

For the benchmark comparison, Microsoft created an application that is precisely functionally equivalent to the Trade 6.1 application, both in terms of user functionality and middle-tier database access, transactional and messaging behavior. This application was created using best-practice programming techniques for .NET and the Microsoft Application Development platform. The resulting application, the ***.NET StockTrader***, is now published on MSDN as a best-practice .NET enterprise application. The .NET StockTrader is a service-oriented application that utilizes Windows Communication Foundation (WCF) for its underlying remoting and messaging architecture. The user interface is an ASP.NET/C# Web application that is equivalent to the Trade 6.1 Java Server Pages (JSP) application. Additionally, the middle-tier services, written in C#, mirror the functionality and transactional characteristics of the backend Trade 6.1 services which are based on J2EE and the IBM WebSphere application server.

Multiple Clients with Open Integration to Middle Tier via WCF

As a service-oriented application based on WCF, multiple interoperability scenarios are enabled with the .NET StockTrader. Since both the J2EE Trade 6.1 and .NET StockTrader applications expose their middle-tier services as industry-standard Web Services, the two applications can be seamlessly integrated with no code changes required. The JSP Trade 6.1 front-end application can fully utilize the .NET middle tier services and messaging capabilities of the .NET StockTrader; and the ASP.NET StockTrader front-end application can fully utilize the EJB-based WebSphere Trade 6.1 middle tier services and messaging capabilities. This interoperability is possible with the .NET StockTrader since WCF, Microsoft's new component remoting and distributed application programming model, is fundamentally based on open Web Service standards including SOAP, XML and the latest WS-* industry standards. In addition to the ASP.NET Web based application that

integrates via services with the middle tier, the sample also includes a Windows Presentation Foundation (WPF) desktop client (also developed in C#), that provides a smart-client interface to the middle tier. The WPF client can also seamlessly connect to either .NET middle tier services, or J2EE Trade 6.1 middle tier services simply by changing the services URL in the configuration page—no code changes are required to achieve this interoperability.

Full Disclosure Notice

The complete source code, all test scripts and all testing methodology for this benchmark are available online. Any reader may download and view the actual code for all implementations tested, and may further perform the benchmark for themselves to verify the results. The benchmark kit can be downloaded from http://msdn.microsoft.com/stocktrader. Extensive time was spent to generate results that represent optimal tuning for the platforms tested, and we are quite confident in the results. We encourage customers to download each kit and perform their own comparative testing and functional and technical reviews of each application.

Tests Performed and Testing Details

Four core benchmark tests were performed:

1. Web Services: Remote activation of backend services from the front-end Web application.
2. Durable Messaging: Orders placed via transacted/durable queue in loosely-coupled architecture.
3. Non-Durable Messaging: Orders placed via a non-durable/non-transacted queue.
4. Monolithic: All elements of application run in a single JVM or CLR instance, no Web Services or messaging. In this mode, orders were set to be placed synchronously.

Both the WebSphere Trade 6.1 and .NET StockTrader can easily be configured to run in each of the four modes above. For each result, we report a peak sustained transaction per second (TPS) throughput rate as averaged over a 30 minute measurement period as tracked by Mercury LoadRunner. Distributed LoadRunner agents drive load against the system tested via simulated Web users running the test script across 40 different distributed client test PCs. Users are added to the system until peak throughput is obtained. Extensive iterative benchmark runs (as required) were done prior to measurement runs to ensure proper tuning of the middle tier systems. For each of the four tests above, we report the peak sustained ***TPS rate***, and also a ***calculated dollar cost per TPS*** so customers can better understand what that performance costs in normalized measurement across systems. The cost calculations are based solely on measuring the middle tier application server software costs (all tests are conducted on the exact same hardware setup). Notes and details on the pricing of the middle tier application servers is included in the Appendix, and based on basic published pricing from each vendor.

Customers should understand that full .NET capabilities are included in every edition of Windows Server, and upgraded versions of .NET are made available for free download from MSDN (for example, .NET 3.0 as tested here). Hence, there is no additional or separate application server cost associated with a .NET application; while commercial J2EE application servers such as WebSphere are separately licensed (and

typically quite expensive) products. The dollar cost per TPS rates might, therefore, surprise some readers.

It is also important to remember that this is a test of specific workloads based on the Trade 6.1 and equivalent .NET StockTrader application. Each application, created by the respective vendors for their platform, however, utilizes most (if not all) of the commonly deployed architectural building blocks used in almost all enterprise applications.

Fair Benchmark Comparisons Between .NET StockTrader and IBM WebSphere Trade 6.1

Since each application supports many different configurations, it is very important to understand what constitutes a fair comparison. First, the configurations compared must be equivalent. This means that the applications must produce the exact same functionality and processing behavior in the configurations compared. You cannot, for example, compare one application running with one-phase transactions between the message queue and the database, while running the other application with a two-phase commit across these distributed resources. The .NET StockTrader, while based on .NET and not J2EE, was designed to mirror most of the Trade 6.1 configurations possible with this testing goal in mind. The key configuration modes Trade 6.1 and .NET StockTrader support, in any combination, are discussed below.

Database Access Technology/Programming Model:

WebSphere Trade 6.1:

- EJB (default)
- Direct

The EJB mode employs a standard IBM/J2EE recommended development paradigm: JSPs invoke stateless session beans, which in turn front-end Entity Beans that use Container Managed Persistence (CMP). The Direct mode eliminates the Entity Beans and CMP, and instead uses direct JDBC calls to the database. Both modes use Java model classes to pass data information between tiers.

.NET StockTrader:

Microsoft has a single data access strategy based on ADO.NET, so there is no mode that equates to this Trade 6.1-configurable setting in the .NET StockTrader application. The ADO.NET implementation uses C# model classes to pass data between tiers. It uses ADO.NET DataReaders isolated in a separate data access layer (DAL) as a best-practice performance programming practice, and to maintain clean separation of database logic from the other tiers of the application.

Interface from Web Application to Backend Business Services:

WebSphere Trade 6.1:

- Standard (via local invocation of EJB Session Beans from JSP pages, not remote/RMI-based invocations)
- Web Services (JSP pages make remote Web calls to the backend services, which do all database/transaction work)

Trade 6.1 Web Services are based on the IBM Web Services/SOAP implementation, which use an Http transport and XML encoding. These are the only possible transport and encodings supported by the IBM Web Services programming model. In Web Services mode, you can optionally direct the endpoint (via their configuration page) to point to any of the .NET Web Service endpoints for StockTrader (discussed below), and the JSP application will seamlessly work with the .NET middle tier services and data access layers.

.NET StockTrader:

- Http_WebService. In this mode, the ASP.NET application makes calls to a Windows-hosted WCF service called .NET StockTrader Business Services. This is a self-host application—self hosting services is a core new concept introduced with WCF, and allows services to be hosted in any application, not just IIS. In this mode, calls are made from the client over HTTP with XML encoding.
- Tcp_WebService. In this mode, the ASP.NET application makes calls to the same self-host WCF Windows application as with Http_WebService. However, WCF enables this same client and same host to work with different "bindings"—in this case TCP as opposed to Http. So in this mode, the application is using **TCP and binary** encoding between the Web application and Business Services. WCF unifies the programming model for *all* remoting in .NET, and separates transport/encoding standards out from the programming logic. So the self-host Business Services program is actually simultaneously supporting Http/XML and Tcp/Binary modes of operation with no extra programming. The WCF clients work the same way, and different clients can simultaneously use different bindings.
- IISHost_WebService. In this mode, the WCF Business Services are hosted in IIS, as opposed to a self-host program. IIS 6.0 hosted services always use Http and XML encoding. However, this is not a restriction for IIS 7.0. For this benchmark, all testing was done on IIS 6.0 however.
- Asmx_WebService. WCF replaced Asmx as the strategic technology from Microsoft to build .NET Web Services. This mode uses classic ASMX 2.0 Web Services, which are always hosted in IIS and only support Http and XML encoding. This is a useful comparison point to understand the performance differences between WCF/.NET 3.0 Web Services and Asmx Web Services.

Order Processing Mode

Trade 6.1

- Synchronous (default). In this mode, orders are simply processed as they come in from the Web tier-- either by way of JDBC logic (Direct mode) or EJB/Entity Bean logic (EJB mode).
- Async OnePhase. In this mode, the application integrates with the IBM "Enterprise Service Bus"—which at its core is their Service Integration Bus (SIB) messaging and message queue

facility integrated into WebSphere. In this mode, a one phase transaction is invoked across the SIB message queue and the database. Orders are lost from the messaging system if the DB transaction fails. No XA/distributed transaction takes place. The integration with the SIB is via a JMS Message-Driven Bean (MDB) bound to a SIB message queue. The SIB message queue can be either in-memory, or configured for persistent storage and "assured message delivery." This means if the application server crashes, the message is still on disk and can be recovered on restart. This is also known as "durable" messaging. However, with a one phase commit, it does not make sense to configure the SIB for assured message delivery, so benchmarks in this mode were configured for "Express Non-Persistent" message queuing for the SIB.

- Async TwoPhase. In this mode, as with the previous mode, JMS is used to integrate with the SIB message queue. In this mode, however, a full two phase distributed transaction takes place when processing orders, so they are not lost if the database transaction fails. Hence, in this mode it makes sense to configure the message queue for persistent storage and "assured message delivery." The transaction is coordinated by WebSphere JTA transaction facilities.

.NET StockTrader

- Sync_InProcess. This mode equates to the Synchronous mode for IBM Trade 6.1. There is no messaging interface, orders are simply processed by Business Services as they come in from the Web application.
- ASync_Msmq_Volatile. This mode equates to running Trade 6.1 with ASync_OnePhase and the SIB queue configured for Express Delivery (non durable, not persisted to disk). In this mode, a WCF service is invoked asynchronously by Business Services to place an order. The WCF Order Processing Service then processes the order on its own. In this mode, the client invokes the WCF service via an MSMQ binding to a non-transacted (non durable) message queue. The message queue is maintained in-memory. This is a loosely coupled mode, however, in that the WCF service host does not need to be running for Business Services (and hence the Web app) to successfully place orders. WCF hides all the MSMQ programming logic from developers---you program message-oriented services in the same way you program synchronous services.
- ASync_Msmq. This mode equates to running Trade 6.1 with the ASync_TwoPhase configuration, with the SIB message queue configured for persistent storage. In this mode, the WCF service is bound to a transacted (durable) MSMQ message queue, and thus presents assured message delivery, since a two phase distributed transaction is always used when processing off the message queue into the database. The transaction is coordinated by the MS Distributed Transaction Coordinator.
- ASync_Http. There is no equivalent Trade 6.1 mode and this mode was not benchmarked. This mode is meant to show how WCF presents a single programming model for a service no matter the transport/encoding standard used. So in this mode, a message queue is not involved; Business Services use an asynchronous Http call to place orders to the WCF host program.
- ASync_Tcp. The same as ASync_Http except over Tcp with binary encoding, not HTTP/XML encoding.

Note for all these modes, you simply configure Business Services, not the Order Processor Host, since the Host is simultaneously listening on all the endpoint types above.

Caching

Trade 6.1

- No Caching (default). The name of this mode is a bit misleading. As long as WebSphere is configured for Servlet Caching, the Market Summary will be cached based on a 3 minute cycle, as defined in the WebSphere Cachespec.xml file. All other elements of the application are not cached, database interactions occur on every request. This is necessary, since the Market Summary query is very heavyweight, and with any real data load, would quickly bring the database and the app to a crawl if run on every visit to the home page. Since the query is the same in the .NET StockTrader application, the same is true for .NET StockTrader. For benchmark runs, we tuned to cache to a 1 minute expiration in both cases since user's would likely want market updates a bit more frequently than every three minutes.
- Distributed Map Caching. This uses IBM's Dynamic Cache Service, based on settings in the Cachespec.xml file, to perform a fairly complex series of caching steps for the application. This enables the application to reduce database calls.
- Command Caching. This is provided for backwards compatibility since the Distributed Map Caching replaces Command Caching as IBM's primary cache technology/recommended approach to caching.

.NET StockTrader

The .NET StockTrader uses the .NET cache API (output caching specifically) to cache the Market Summary page for 60 seconds. We chose not to implement further caching in the application because it is simply not realistic. While Trade 6.1 can cache stock quotes, account data, portfolio data and the like (and .NET StockTrader could too if implemented), the simple fact is that this is not a realistic approach or "cache policy" for this application. Consider that the IBM cache, while distributed (it can keep cached items in sync across clustered servers), is not invalidated by the data source itself. Hence, an update to a database table by any other application using the same database would result in possibly corrupt data, or at least presenting incorrect "stale" data to users in the application. Unless a customer is willing to direct ***all*** database updates/deletes/inserts through the Trade 6.1 entity beans, data on the middle tier would quickly become out of sync with the actual database. In other words, such a strategy keeps the organization from building new applications against a common database. Market Summary information is fine to cache, since its read-only and can stand to be 60 seconds stale; user account balances and stock price information used on trades cannot. Hence, the .NET StockTrader does not implement further caching beyond Market Summary.

It should be noted, however, that Microsoft introduced a new SQL Cache Dependency feature for the .NET cache with .NET 2.0; and unlike IBM's Distributed Cache technology, this is directly invalidated by the data source itself. In other words, a completely separate application (or even

a manual update to a single row of data) will cause and application cache that contains that row to be invalidated, such that data will remain in sync with the database itself. While this is a great feature, it is not appropriate for constantly changing data (since the subscriptions generate network traffic). Such constantly changing data should likely not be cached to begin with. Given the nature of the benchmark, almost all data is constantly changing in the benchmark runs, so again, for realism we do not use this .NET feature.

Enable Long Run Support

Trade 6.1

- On (default)
- Off

This setting was introduced to Trade with WebSphere Trade 6.1, apparently because customers running the benchmark for long periods of time saw steady degradation of performance as user accounts started to contain more and more orders. This resulted in heavier queries, and large and steadily increasing amounts of data being passed between tiers and formatted into the account summary page. Hence, IBM created this setting. All it does is completely eliminate the recent order query and order display on the account page. It is understandable that in a benchmark setting, you cannot control the amount of data as you would in a production application (typically, for example, by querying based on order date, etc. for a restricted set of data). However, there are better approaches to solve this problem. For example, capping the maximum orders returned by a given query at a reasonable number (like last 100 orders). EJBSQL, however, does not support this capability, so IBM chose this option instead.

Despite this discussion, there is a much better way to ensure benchmarks can be run for long periods of time using Trade 6.1 and StockTrader---and a solution that makes the benchmark more realistic at the same time. That solution is to use a much larger data load in the database. For example, with 500 users, and a benchmark script creating 100 orders per second, it would take just 5000 seconds (83 minutes) for each visit to the account page to cause a 1000 record SQL query to be run on each request, followed by passing 1000 records over the network for each user on each request. This is the default load IBM uses for the database. We chose simply to increase the default data load, which also makes the benchmark more realistic---very few enterprise production databases would be so small. We used 500,000 accounts, each with 5 existing orders (2.5 million orders). Now, at 100/orders per second during a benchmark, it would take 58 days to get to 1000 orders per user.

.NET StockTrader

As mentioned, there is no equivalent setting for StockTrader, instead we run the benchmark for both applications against a larger, much more realistic data load (500,000 accounts, 5 order per account, 100,000 quotes). We provide a database loader (written in .NET Windows Forms) to load both SQL Server 2005 and DB2 V9. This loader runs significantly faster than the Trade 6.1 JSP Data Load Page. This loader program will also reset the database between benchmark runs to the same starting state by deleting added data records.

In summary, you should make sure to de-select "Enable Long Run Support" in Trade 6.1 before doing benchmark comparisons involving the account page, since this only disables the account page functionality which is not realistic.

Database Load

As discussed, we used a default load for all benchmark runs (reset between runs) of 500,000 accounts, 5 orders/holdings per account, and 100,000 quotes. This is much more realistic than the IBM default settings.

Database Configuration

The IBM Trade 6.1 application was tested against an all-IBM configuration, using IBM DB2 V9 (Enterprise Edition) as the backend database, and the latest IBM DB2 V9 JDBC drivers for data access. The .NET StockTrader was tested against an all-Microsoft setup, with a backend SQL Server 2005 database (Enterprise Edition). The benchmark is not a database benchmark: enough capacity was employed for the database hardware to ensure it was not a bottleneck in any benchmark run. Each database was deployed to the same 64-bit Windows Server machine, running on a 4-processor 2.2 GHz AMD Optereon system. The database was configured with two fast RAID arrays (15 ms disk access times, 14 drives each in a RAID 10 configuration); logging was directed to one array, the primary database files were stored on the second array. Each array was configured with its own dedicated controller. The 64-bit editions of DB2 and SQL Server were installed. All tuning steps were followed for DB2 according to the Trade 6.1 documentation, however, additional logging space was configured given the larger data load. The equivalent drive space was configured for SQL Server logging. The database disk usage was closely monitored to ensure each run could complete without requiring the database to extend the logging or data file space during a benchmark run. This is an important consideration for custom tests.

Test Scripts

Mercury LoadRunner was used to record test scripts—browser interactions that exercise most of the functionality in the application. These were run across 40 client machines (500 MHz Windows XP desktops with 512 MB RAM). User agents were configured to run with a one second think time between each request. Each benchmark run included a warm up run to get to steady state, and a 30 minute measurement period. TPS rates were determined by LoadRunner by averaging across the 30 minutes. Error rates were monitored to ensure they remained at less than .01% during the measurement period. Some dropped connections and or database deadlock conditions do result from running very large user loads (great than 1,000 concurrent users at a one second think time) against the applications. Dropped connections, if any, occurred only during the warm-up period as larger user loads were ramped against the applications. User loads were run for each application up to a number that represented peak throughput for that configuration, as determined in many iterative runs (literally hundreds) during the tuning stages. Extensive time was spent tuning IBM WebSphere (see the appendix) to achieve peak throughput for the software/hardware configuration tested. IBM does not publish pre-set tuning guides for Trade 6.1, and developers must iteratively test and tune the various knobs in WebSphere (there are many) to get to an optimal setup for a given hardware configuration and software workload. .NET does not require nearly as much tuning, and in general will scale quite well out of the box, given a properly

coded application. Some tuning was applied, however, and this is documented in detail along with the WebSphere and Linux tuning in the appendix.

In general, it is fairly straightforward to recognize an in-properly tuned system for both platforms. Given enough database capacity, each application server running under load should be able to reach full saturation (~100% CPU saturation) when properly tuned. Just because the system can reach full saturation, however, does not mean the tuning is optimal, it just means if it can't reach this, then it is not properly tuned. Hence it is a requirement to iteratively test after adjusting, individually, the core tuning settings for the application. With Trade 6.1, these are extensive, including several different thread pools, connection pools for databases and queue connection factories, Java heap sizes, and the like. Several man-months were spent on this effort, and the WebSphere installation was always kept up-to-date width the latest Refresh Packs from IBM during testing. We did notice a good increase in performance in several scenarios between WebSphere 6.0 and 6.1. We are quite confident in the results and the system tuning applied, however, should IBM recommend different (specific) settings, we will be happy to re-run the benchmark and re-publish new results. Customers can also run the benchmark—it is a great way to really judge the capacity of the two platforms for a realistic workload, and then to judge the cost of each platform and compare the cost to the results achieved.

Simulated User Settings

Mercury agents were set to not download images (this is not a web server/network I/O benchmark) during runs. They make requests to the application server, and all processing is completed and just the HTML returned to the agent. This reduces the overhead on the agent machines, and helps ensure the 40 client machines used in the testing never become an artificial bottleneck. Just as importantly, the agents were **configured to reset connections between iterations**, to simulate constantly new users logging into the application, and more fully exercise the underlying networking stacks and HTTP keep alive system that the application servers and Web servers use to support large concurrent user bases. Too many benchmarks are simply run with 10-15 threads, no think times and no network resets between script iterations. These types of benchmarks often produce very different (often over-inflated) results than real world usage conditions. Our settings are meant to much more closely mimic the real world. Think times (even if just one second) and connection resets between iterations make all the difference here.

Changes to the IBM Downloadable Version of Trade 6.1 as Used for Testing

We wanted to avoid making changes to IBM's code; after all, it was developed by IBM for their own platform as a best-practice performance application for performance testing and capacity planning. The only setting we changed, therefore, besides ensuring the cachespec.xml file was only caching Market Summary as did the .NET application (as previously discussed), was to ensure the application did not make requests to the Stock Streamer sample Java Client application published with Trade 6.1. This application uses a Topic-based pub/sub mechanism to show a subset of stock trades at periodic intervals in a Java client application. Since we did not implement (at least not yet) this equivalent functionality for .NET StockTrader (we instead did a full blown WPF client); we needed to make sure the Trade 6.1 application was not making JMS calls for each stock trade. This is accomplished easily by merely changing environment entries (PublishQuotePriceChanges) in the deployment descriptors that disable

this functionality specifically, or by merely commenting out the call to publish to the JMS topic for the Streamer application. Correct settings can be ensured by turning on Trade 6.1 detailed logging, and ensuring these calls are not being made (which we did). Of course logging should be turned off for the actual benchmark. Note however, that updates to stock prices and volumes remain on (UpdateQuotePriceVolume), as the .NET StockTrader also updates stock prices and trading volumes in the equivalent fashion for each order placed as part of the buy and sell transactions.

Security Settings

Trade 6.1 does not implement any security between the Web application and the JSP engine—for example, no encryption of passwords takes place. We made sure the .NET StockTrader similarly did not implement any extra security, although this would be appropriate for a Web-based deployment. ASP.NET Forms Authentication was used as the authentication mechanism (with anonymous Internet access ala Trade 6.1), and ASP.NET Forms Auth makes it extremely easy to implement any number of encryption algorithms simply via a configuration setting. The application, if ever deployed on the Web vs. an Intranet, would require SSL/HTTPS as the primary security mechanism. Customers can configure IBM HTTP server or IIS for SSL optionally for additional testing if they desire. The Web Services in the application as implemented by IBM are simply SOAP 1.1 based. They do not employ WS-*. Therefore, neither does the 1.0 implementation of StockTrader, but the applications present a possible way for benchmarking of various WS-* standards in the future, such as WS-Atomic transactions and WS-Reliable Messaging which WCF and .NET fully support. It should be noted, however, as the applications do not require federated security, in the real world neither would likely implement WS-Security for an actual deployment, considering the overhead and lack of need for it in this specific application as designed.

IBM HTTP Server vs. Port 9080

When benchmarking IBM WebSphere, it is important to understand that while WebSphere provides an in-process HTTP listener service (port 9080, by default), IBM best practice recommended deployments are in conjunction with the full-blown IBM HTTP Server (a repackaged version of Apache). Hence, for all configurations, we used IBM HTTP Server as packaged with WebSphere, configured with the WebSphere Plugin. This includes the distributed Web Service benchmark runs, where four client application servers execute the JSP Web application (co-located with the IBM HTTP Server); and make requests to the Web Service Host application server. These requests are made on port 80 to the IBM HTTP Server that is installed on the WebSphere Web Service Host application server. This is the recommended IBM configuration for deploying high-load application servers hosting Web Services. IBM HTTP Server tuning details for Windows and Linux are included in the Appendix.

Web Application Pages Exercised by the Test Scripts

The test scripts were designed to drive load on the system in a way that exercises most functionality in the application, and puts a heavier emphasis on transactions; such as adding new registered users and buying stocks. The precise flow of the test scripts (exactly the same for all test runs on all platforms) is listed below. A one second think time (smaller than real-world, to driving more load per simulated user) was placed between all URL requests, and in the results, a ‘transaction’ is defined as the successful completion of the URL request to the server, with valid response/HTML returned.

- Login random registered user (1 to 500,000 users loaded in database; the login includes in both apps a redirect to the home page, and all the logic to login and display home page)
- Request four random quotes (1 to 100,000 distinct quotes loaded in database; one post performed with 4 stocks requested)
- Request four random quotes (1 to 100,000 distinct quotes loaded in database; one post performed with 4 stocks requested)
- Visit Portfolio Page
- Visit Account Page (no account update performed)
- Visit home Page
- Logout the Registered user via logout page
- Register a new user/submit registration form (this also logs new user in with redirect/display of home page)
- Visit Portfolio Page
- Buy a random stock symbol (1 to 100,000 stock symbols in database; buy operation involves a direct post/submit to the order submission pages, which submit the order for all backend processing)
- Visit Home Page
- Buy a random stock
- Visit Account Page
- Get quotes for 4 random stocks (one post performed with 4 stocks requested)
- Buy a random stock
- Buy a random stock
- Visit Portfolio Page
- Visit Home Page
- Logout

32-Bit versus 64-Bit Testing

Middle Tier

For this benchmark, we could test either the 32-bit versions of .NET and IBM WebSphere, or the 64-bit versions. Based on published documentation from IBM on 64-bit comparisons (see ftp://ftp.software.ibm.com/software/webserver/appserv/was/64bitPerf.pdf[1]), including the Trade 6.1 benchmark, and pre-verification runs of the Trade 6.1 application on 64-bit WebSphere on both

[1] Note that you should not directly compare the results for Trade 6.1 workloads in this referenced benchmark paper to the results obtained here. IBM does not report the test scripts used to generate the results (what operations are performed?); or the test tool used to generate the results. For example, Trade 6.1 ships with a built-in JSP workload driver test tool that can generate benchmark results that was likely used. However, this does not exercise the application in the same way as our Mercury test scripts, which are more intensive in terms of the % of orders and visits to Portfolio, Home and Account pages (vs. just retrieving stock quotes). They also do not report the caching modes or whether the "Long Run" option, which disables the Account Page functionality when turned on, was on (default) or off.

Windows and Red Hat Linux/Opteron, we determined the Trade 6.1 application performs roughly 10-20% slower on 64-bit WebSphere. The same is true for the .NET StockTrader, which also performs roughly 10-20% slower on 64-bit Windows/.NET vs. 32-bit Windows/.NET. Both IBM and Microsoft have documented that unless a middle tier application requires more than 2GB of addressable address space (for caching, for example), or the application performs math-intensive floating point operations (as required with heavy use of encryption, graphics engines, databases, etc.), typical middle-tier business services will perform slightly better on their respective 32-bit platforms vs. 64-bit platforms given the same hardware system. Why? There is extra overhead for the Java and .NET runtimes to carry around 64-bit memory addresses. For applications that will not benefit from the faster math calculations or the ability to address more memory than 2GB per process, the extra overhead can decrease performance on 64-bit software platforms vs. 32-bit software platforms. Neither Trade 6.1 or .NET StockTrader benefit from using their respective 64-bit platforms (.NET/WAS). We chose to run the middle tier on the fastest setup of the hardware tested, therefore: 32-bit .NET and WAS running on the 4 x 1.8 GHz Opteron system under test. For a different 64-bit benchmark, inclusive of WebSphere and .NET results, and a discussion of 64-bit .NET on the middle tier, please also refer to http://msdn2.microsoft.com/en-us/vstudio/aa700838.aspx.

Database

The database, however, which can benefit enormously from the ability to directly address > 2GB of RAM (especially since we used a good-sized database load for the tests), was run on 64-bit Windows Server running the 64-bit versions of DB2 V9 and SQL Server 2005. This computer was configured with 16GB of RAM.

As published benchmark kits, the two applications do present the chance for customers to run their own tests, which we highly encourage. For example, running under an SSL configuration may see better results with a 64-bit middle tier vs. a 32-bit middle tier. Such testing was beyond the scope of this set of benchmark tests.

Benchmark Results

Web Service Benchmark

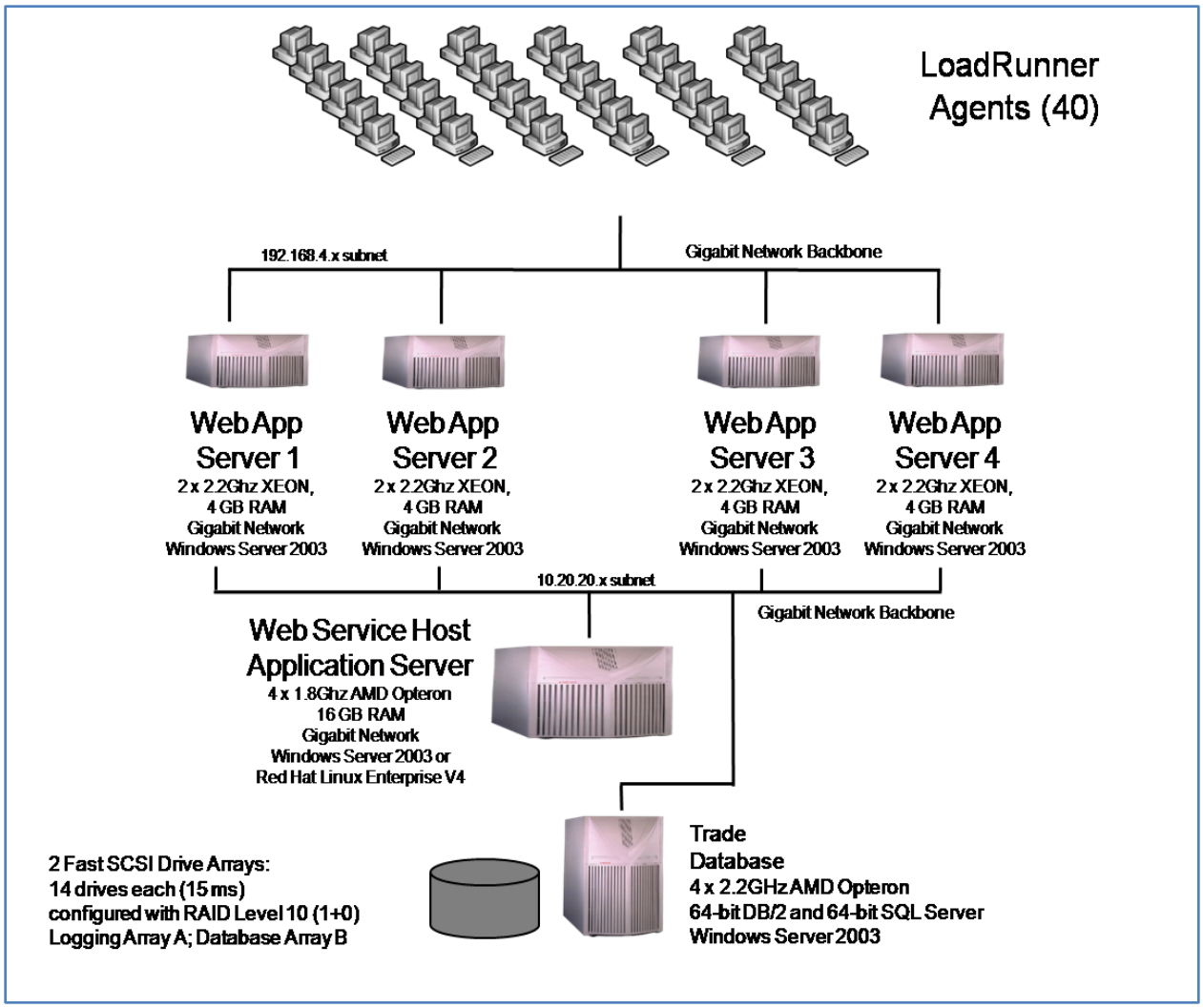

Figure 1: Benchmark Test Bed for Web Service Remote Tests. The setup ensures that the Web Service Host is the System Under Test (SUT); as neither the 4 Web Application Servers running the Web application or the database are near capacity during any benchmark run. Hence, we are measuring the throughput of the Web Service Host application server in all tests.

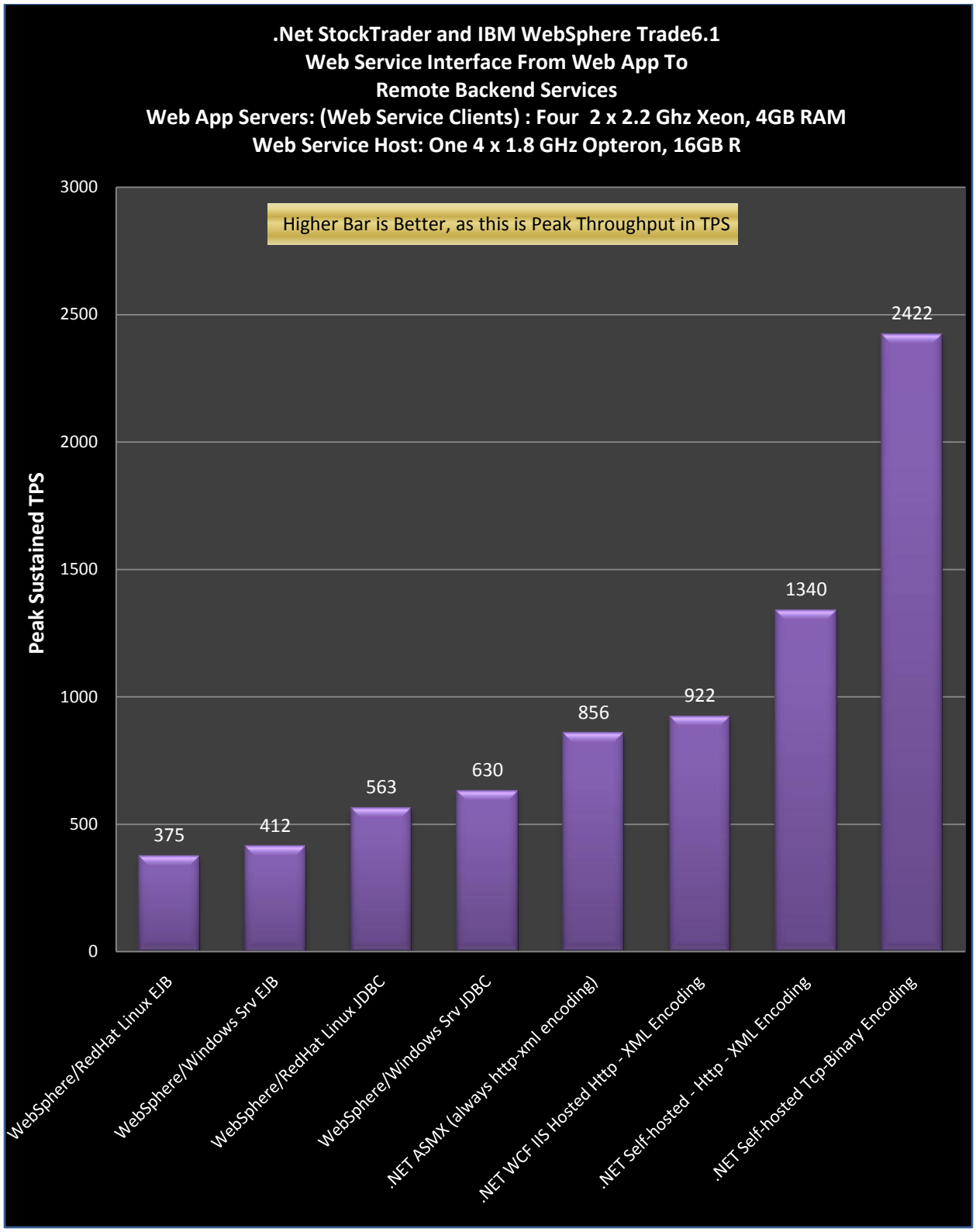

Figure 2: Peak TPS Rates for the Web Service test

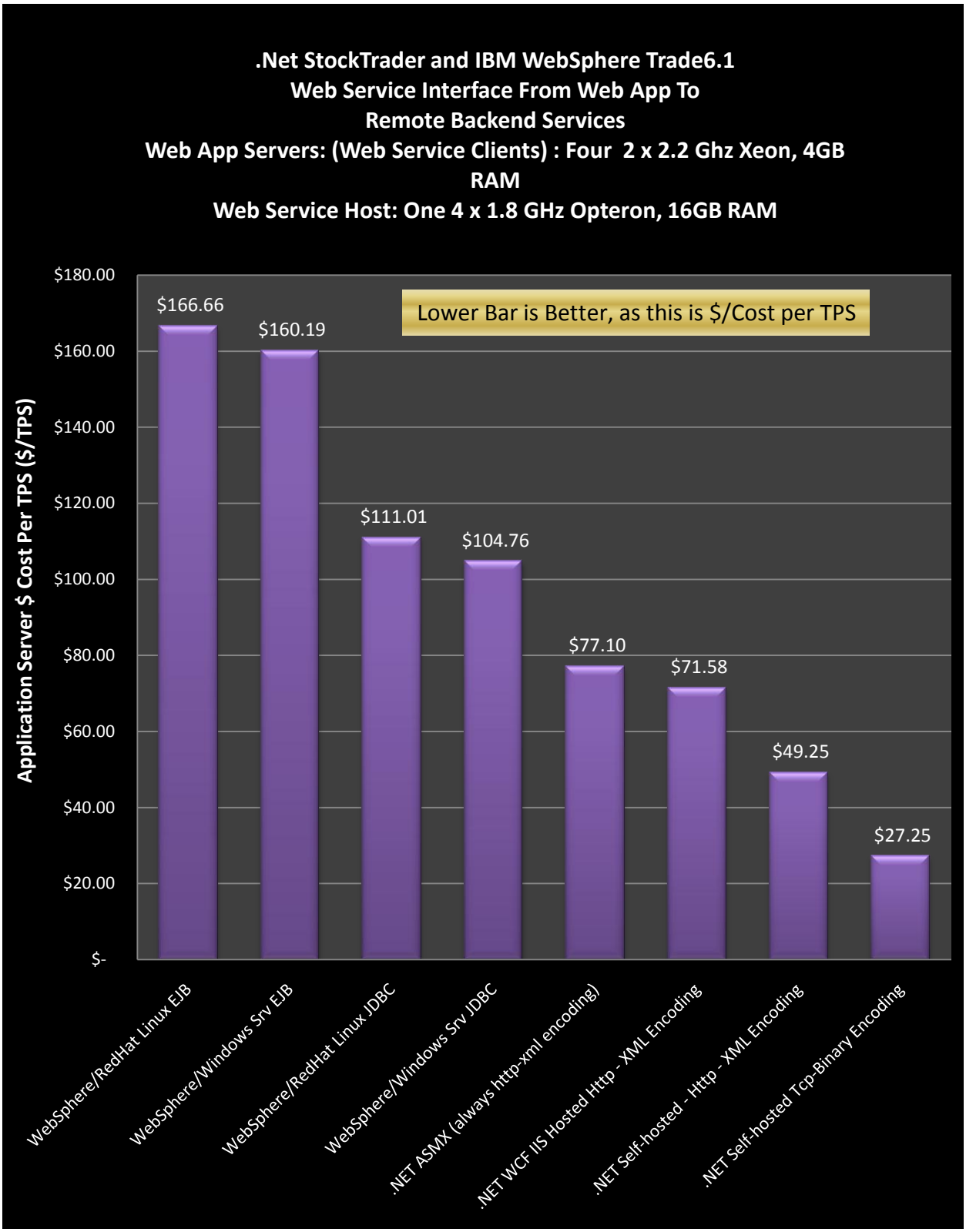

Figure 3: Price/Performance chart for the Web Service Test. Refer to Appendix A for pricing calculations for the middle tier software.

The Web Services Benchmark Discussion

In this test, the Web Application servers (running the JSP or ASP.NET front ends) are the Web Service client machines, using Web Services to remotely invoke the service layer, hosted on the Web Service Host application server. Both the clients and the host are performing XML serialization and de-serialization. The Web Application server clients are accessing the services via a SOAP Proxy (Trade 6.1), or the WCF client (.NET StockTrader). We set the test up with four distributed Web Application Servers, with round-robin load balancing performed from the Mercury Controller as iterations are performed, such that all four Web Application Servers get equal load. Each in turn makes remote network requests to the Web Service Host, which is servicing all four Web Application Servers. The use of 4 Web Application Servers ensures we do not have a bottleneck on the Web Tier, and we are accurately comparing just the performance of the Web Service Host computer for all benchmark runs. Some important conclusions can be drawn from this test:

1. Both the ASMX and WCF SOAP/HTTP configurations are significantly faster than IBM WebSphere 6.1 for hosting Web Services. .NET WCF hosted in IIS offers 46% better throughput than the JDBC WebSphere configuration in this test; and 124% better throughput than the EJB implementation.
2. Self-hosting WCF services can lead to performance advantages over hosting .NET Web Services in IIS—even when operating over an HTTP-XML basicHttpBinding. Self hosted WCF HTTP Web Services offer 56% better throughout than the ASMX equivalent services hosted in IIS for this test. They offer 45% better throughput than the equivalent WCF IIS-hosted service operating over Http-XML.
3. Self hosted Web Services using WCF significantly outperform IBM WebSphere Web Services. The self-hosted WCF services operating over Http-XML (full SOAP compliance) offer 113% better throughput than the fastest WebSphere Web Service results (JDBC data access). They offer 225% better throughout than the WebSphere Web Services using EJB entity bean database access.
4. The self-hosted WCF services can also support, simultaneously, the netTcp WCF binding, with binary encoding. This can lead to significant performance boosts for remote calls. The WCF netTcpBinding replaces .NET Binary Remoting (used with .NET 1.1 and 2.0) as the preferred way for remote calls between .NET clients and remote .NET services. Supporting both HTTP/XML and TCP/Binary requires no extra development, as WCF unifies the programming model for HTTP-based Web Services and .NET Binary-remoted components, and service hosts will listen simultaneously on all configured endpoints to support any different type of client on any platform.
5. The Tcp-Binary binding (netTcpBinding) between the ASP.NET clients and the Web Service host offer 81% better throughput than the WCF basicHttpBinding used in the same self-host .NET executable. The Tcp-Binary remote mode offers 284% better throughput than the fastest WebSphere Web Service configuration (JDBC data access).

Refer to the appendix for the performance monitor captures for this test taken after initial warm-up at steady-state throughput rates for each configuration.

Messaging Benchmark - Durable Queue with Two Phase/Distributed Transactions

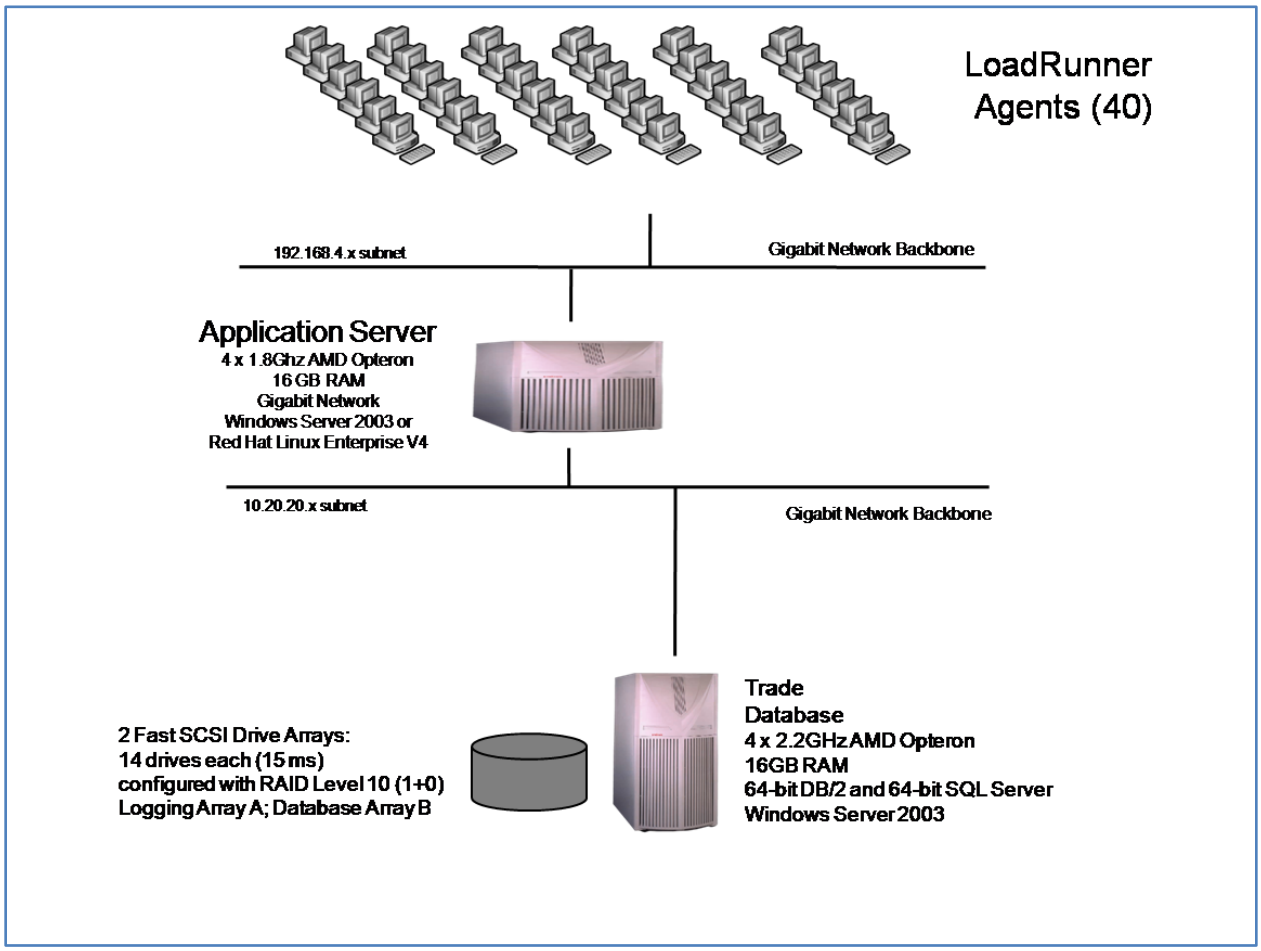

Figure 4: Figure 1: Benchmark Test Bed for Messaging Tests. The Messaging engine (MSMQ or IBM's Service Integration Bus Message Queue/JMS messaging engine) are co-located on the same application server as the other parts of the application in this test. Benchmark agents drive load against the Web application, which in turn accesses the Trade Services running in-process (the same CLR or JVM instance). The Trade Services then place asynchronous orders via the Trade 6.1 JMS Broker MDB, or the .NET StockTrader WCF Order Processor Service.

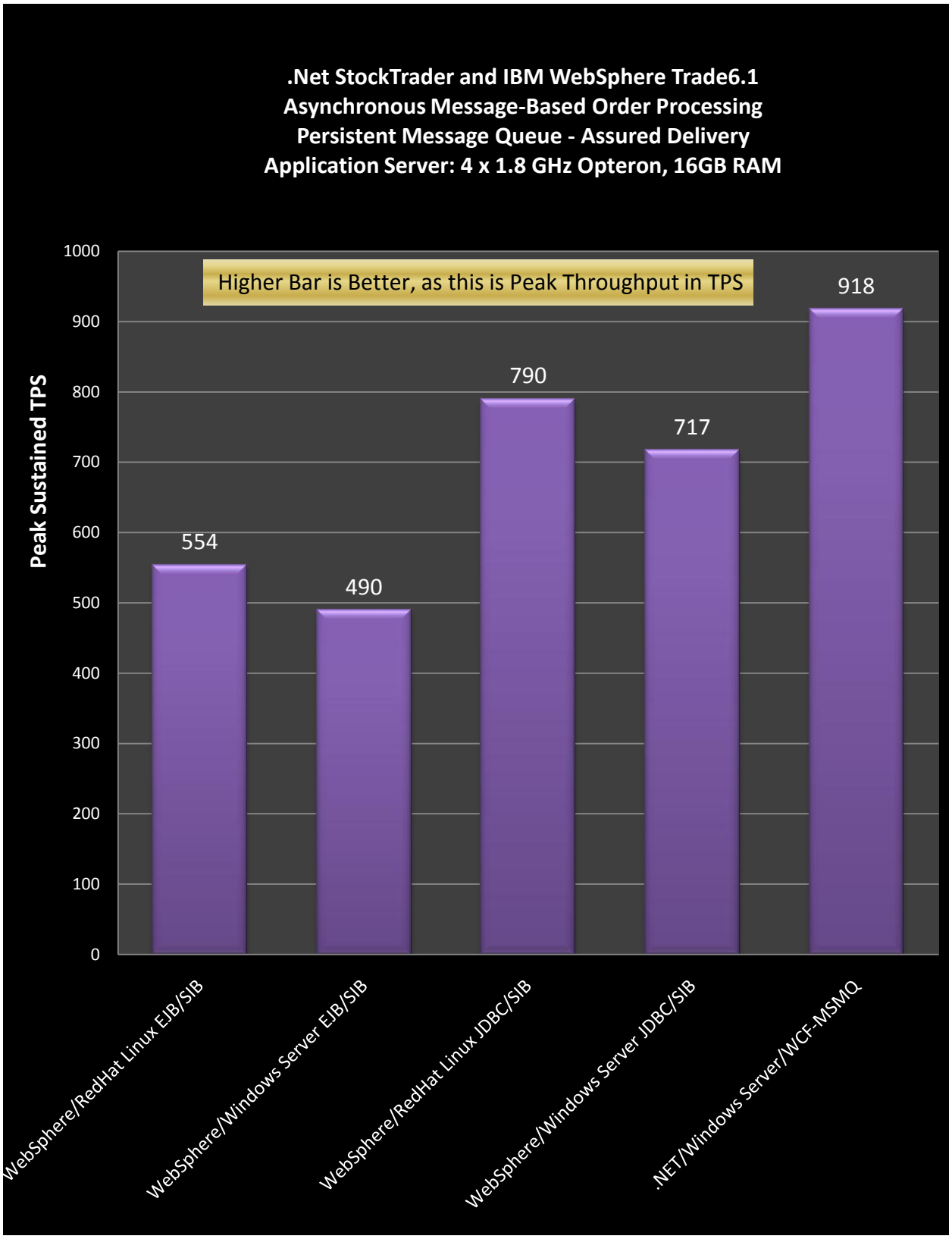

Figure 5: Peak TPS Rates for the Durable Messaging Test

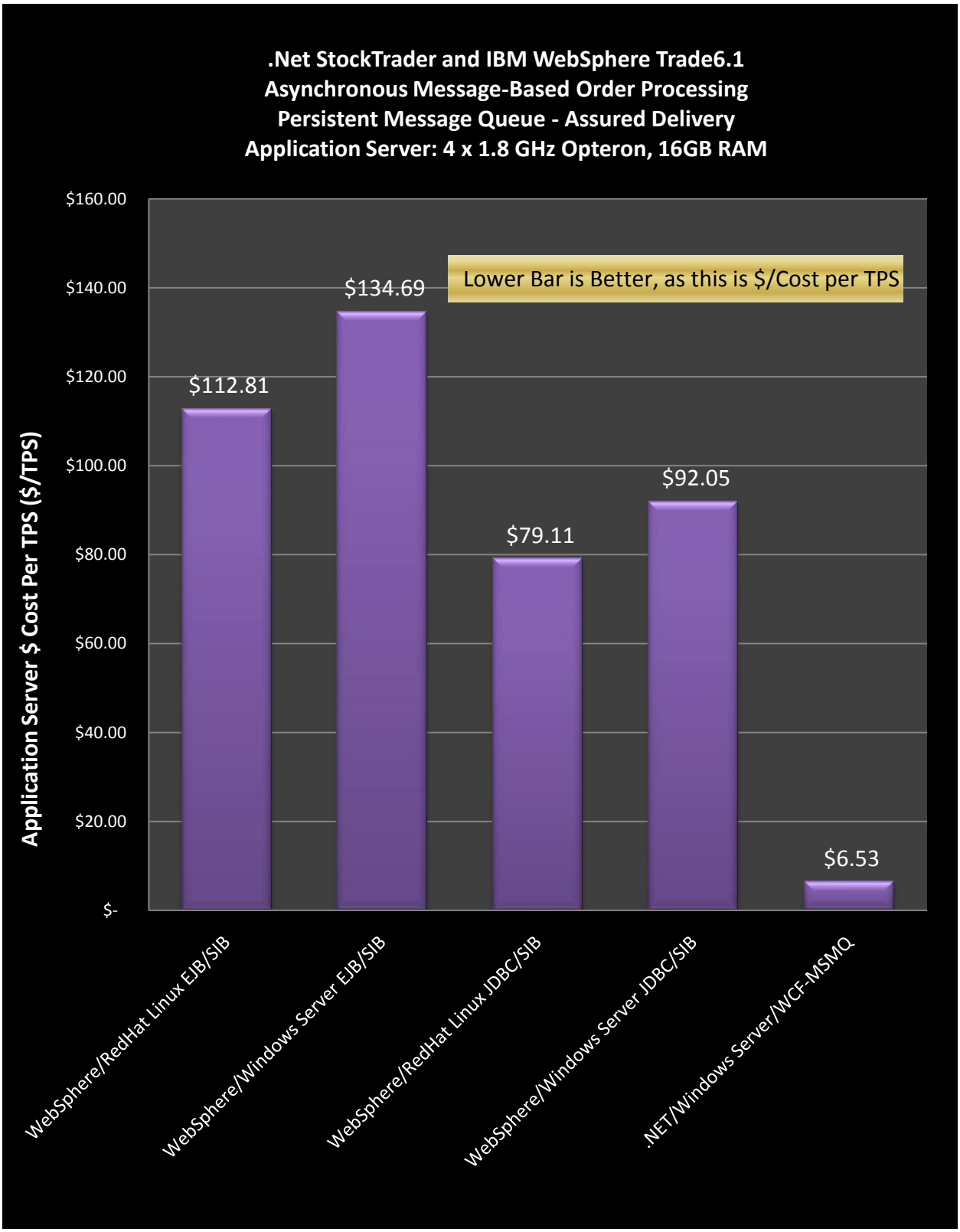

Figure 6: Price/Performance Chart for the Durable Messaging Test. to Appendix A for pricing calculations for the middle tier software.

Messaging Benchmark Discussion – Durable/Persistent Message Queue

In this test, the OrderMode is set to Asynchronous-TwoPhase for Trade 6.1, with the SIB queue configured for persistent storage and Assured Message Delivery. This is the configuration that would be used in a production application, with the reads from the queue and the corresponding database inserts/updates occurring as part of a single atomic, distributed transaction. This ensures that messages are not lost if a database processing failure occurs. For .NET StockTrader, the OrderMode is set to ASync_Msmq. In this mode, the WCF Service is bound to a transacted (durable/persisted) message queue, with a similar two phase distributed transaction when processing orders off the queue. Again, this mode ensures messages are not lost if a database processing error occurs. Some conclusions that can be drawn from this test:

1. Again, as in all tests, the JDBC "Direct" mode for WebSphere offers better performance than the use of EJB entity beans.
2. .NET WCF outperforms the WebSphere EJB configuration in this test by 66%; it outperforms the JDBC configuration by 16%.

Messaging Benchmark – Non-Durable Queue with One Phase Transactions

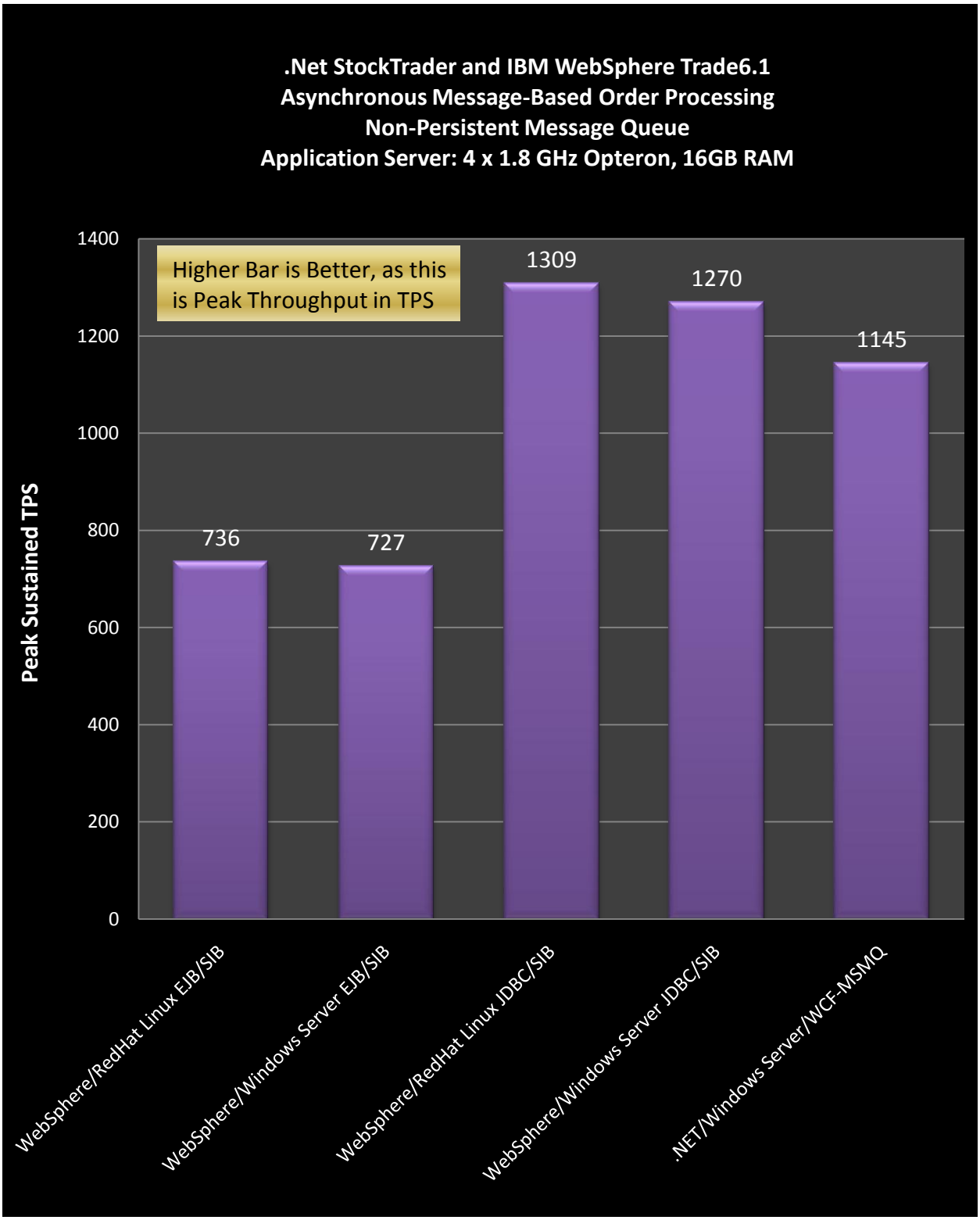

Figure 7: Peak TPS Rates for the Non-Durable Messaging Test

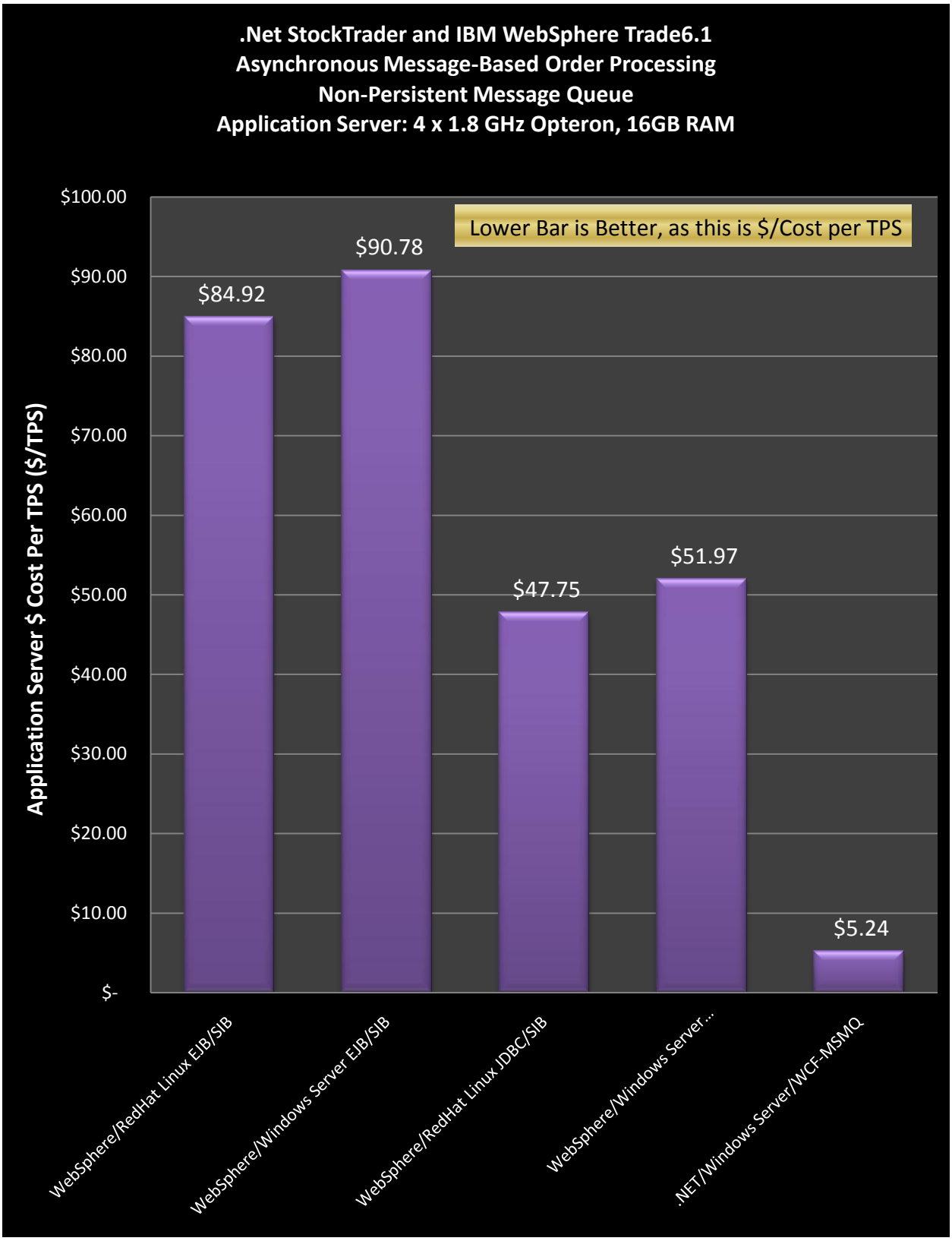

Figure 8: Price/Performance Chart for the Non-Durable Messaging Test. Refer to Appendix A for pricing calculations for the middle tier software.

Messaging Benchmark Discussion – Non-Durable/Non-Persistent Message Queue

In this test, the OrderMode is set to Asynchronous-OnePhase for Trade 6.1, with the SIB queue configured for express non-persistent storage. For .NET StockTrader, the OrderMode is set to ASync_Msmq_Volatile. In this mode, the WCF Service is bound to a non-transacted (in-memory) message queue, with a similar one phase transaction when processing orders off the queue. While this configuration would likely never be used in a production application, it is included here for completeness. Messages will be lost in this configuration in two cases:

- The application server/messaging engine crashes.
- There is a database processing error of some sort, after the message is read from the queue.

Some conclusions that can be drawn from this test:

1. Interestingly, in this mode WebSphere outperforms .NET and WCF. One potential explanation is that the WCF self-host program is run out-of-process with respect to the Web application; while with WebSphere, the JMS engine is run in the same JVM process as the Web application. In fact, in the WebSphere configuration, only a single JVM process is involved in an order operation—so no process hops are involved. In the .NET tests, the Web application runs in one process (ASP.NET worker CLR process), the WCF self-host runs in another CLR process, and the messaging engine (MSMQ Service) runs in a third process. While this is true for the .NET durable/persisted runs as well, in the non-persisted/one phase tests WebSphere does not have to process a distributed transaction or persist messages to disk. While .NET may be much faster in these key operations, when they are not part of the benchmark test, that advantage is lost; while at the same time WebSphere benefits from running everything within one JVM instance/process. Running the messaging engine and core application in the same instance (default for a Trade 6.1 install) is not as reliable, of course, as separating these components into separate processes. To be fair, however, with extra configuration and setup, WebSphere could run multiple Application Server instances on the same machine and run the messaging engine in a separate process from the core application; such a setup is beyond the scope of this benchmark.
2. Keep in mind this configuration is really not relevant for a production application that must have reliability guarantees on message processing.

Data-Driven Monolithic Web Application Benchmark

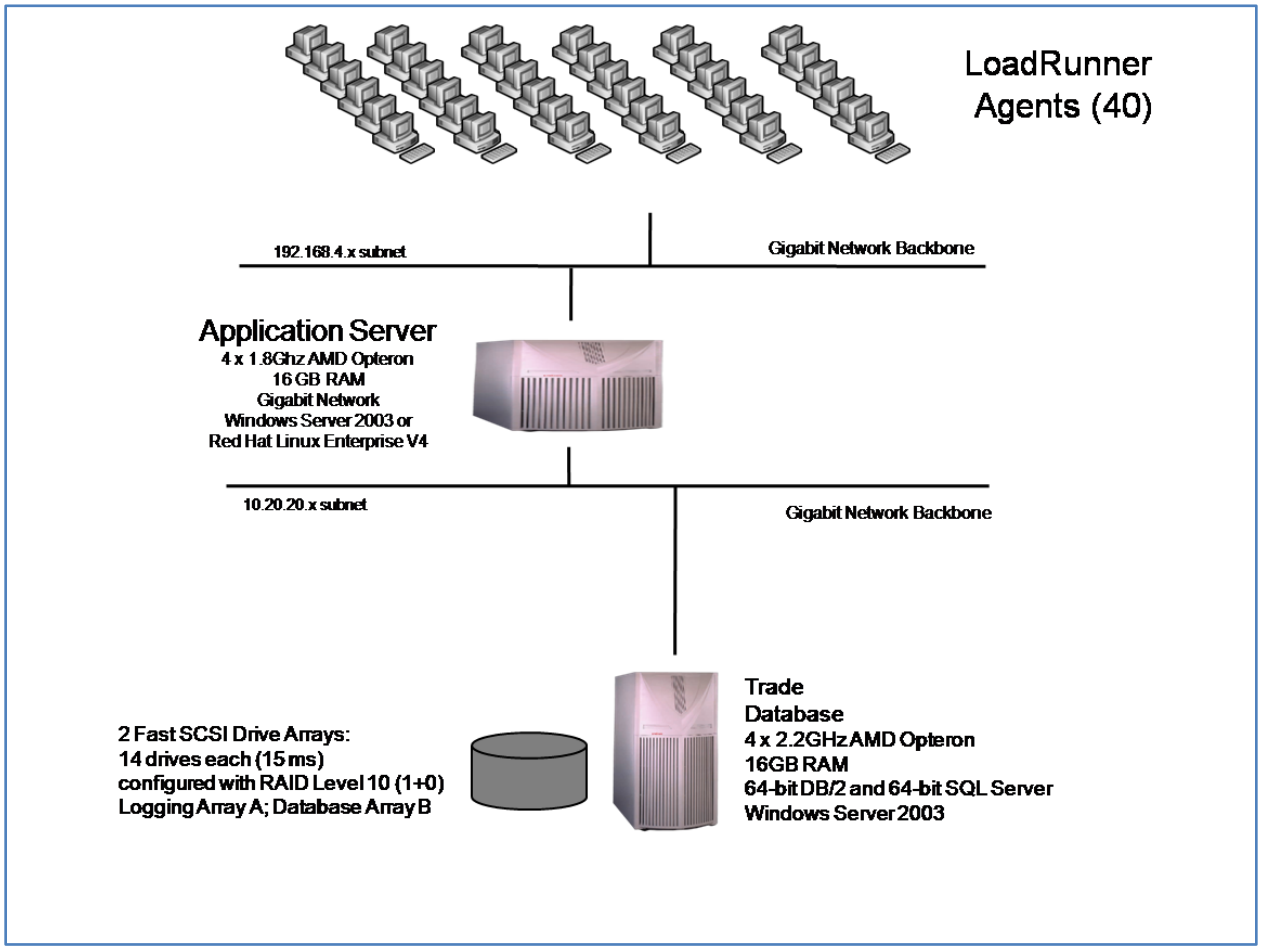

Figure 9: Test Bed setup for the monolithic Web application benchmark. In this setup, which is the same physical setup as the messaging tests, orders are placed synchronously. There are no remote calls made, and no JMS or MSMQ messaging is involved as all orders are placed synchronously. Since a WCF Service host is not involved for messaging or remoting, all elements of the .NET StockTrader run within the ASP.NET worker process (a single CLR instance); just as the Trade 6.1 application runs in a single JVM instance.

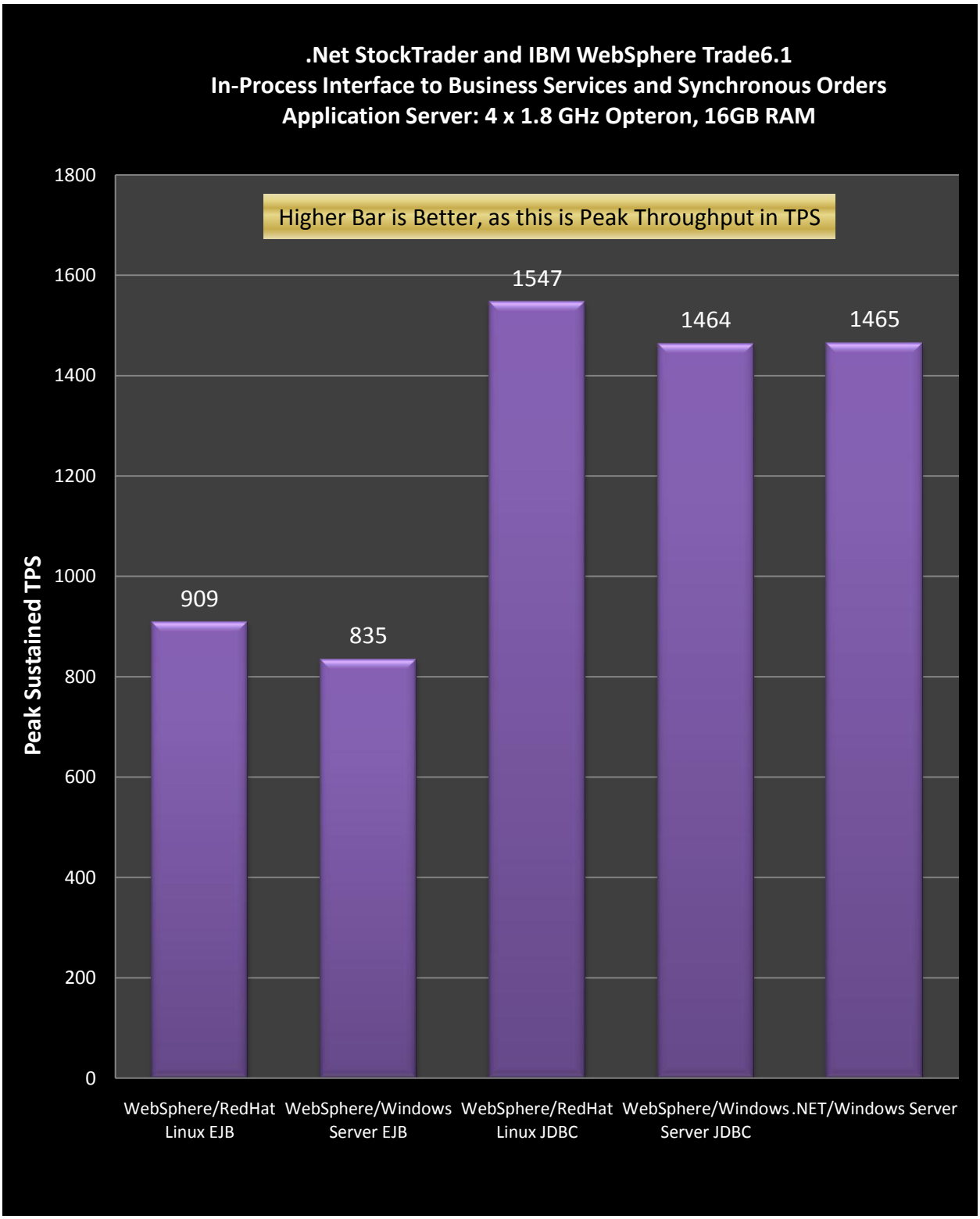

Figure 10: Peak TPS Rates for the Monolithic Web Application Test.

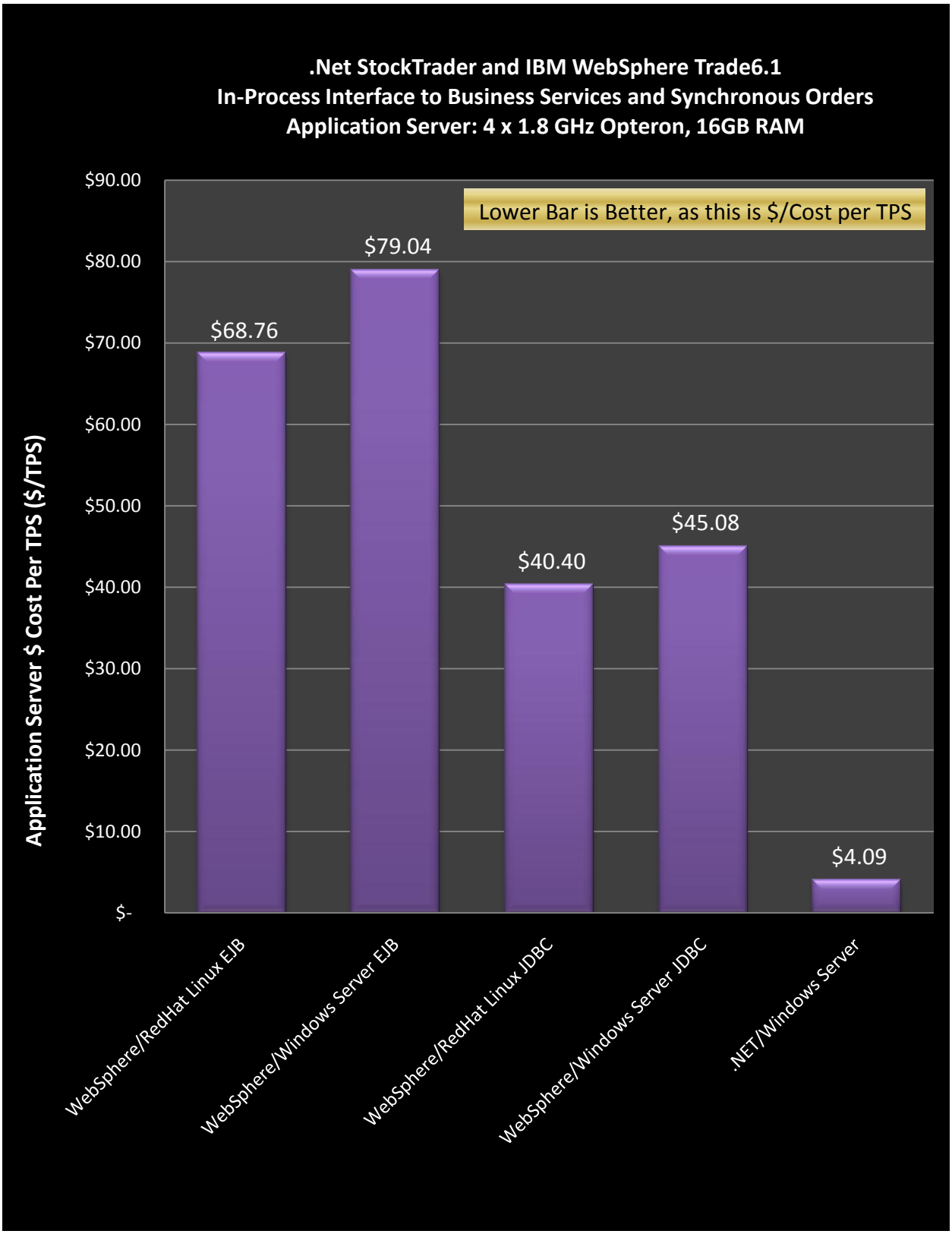

Figure 11: Price/Performance Chart for the Monolithic Web Application Test. Refer to Appendix A for pricing calculations for the middle tier software.

Data-Driven Monolithic Web Application Benchmark Discussion

This mode of operation represents a non-service oriented, monolithic application. As such, there is no service-reuse possible, and no ability to "plug in" different clients or applications/services running on different platforms. The performance is quite good, considering there are no distributed transactions, no messaging/queuing, and no remote calls whatsoever between the Web Applications and the middle tier processing components. All elements of the application must be deployed in unison, and hence versioned/updated in unison. Nevertheless, for applications that do not require Web Services, remote calls, or messaging, this mode of operation, for both applications, is a viable choice for a deployment, and one which can provide very fast performance.

Application Architecture Diagrams

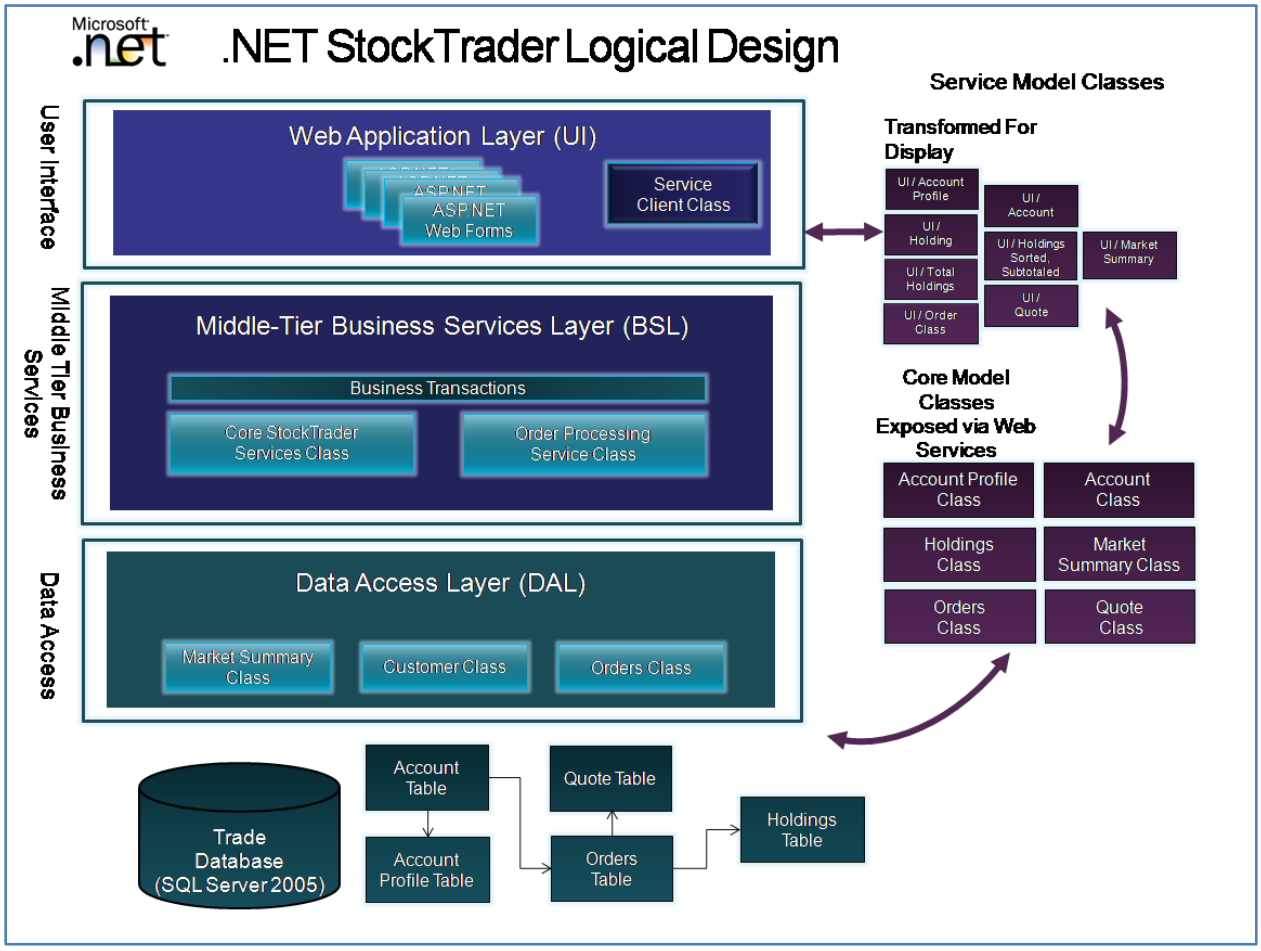

Figure 12: Logical Design of .NET StockTrader

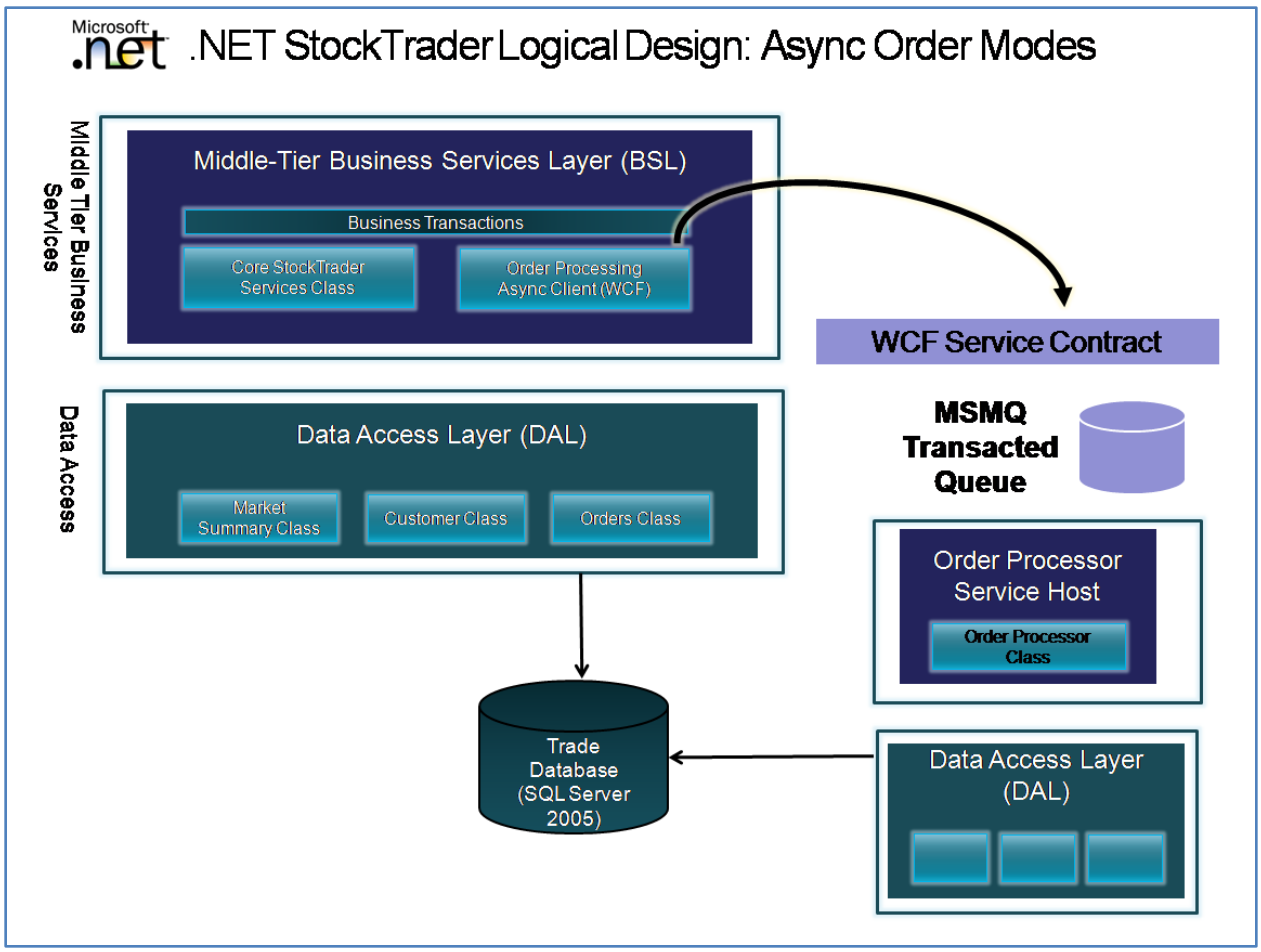

Figure 13: Asynchronous, Message-Oriented Processing of Orders

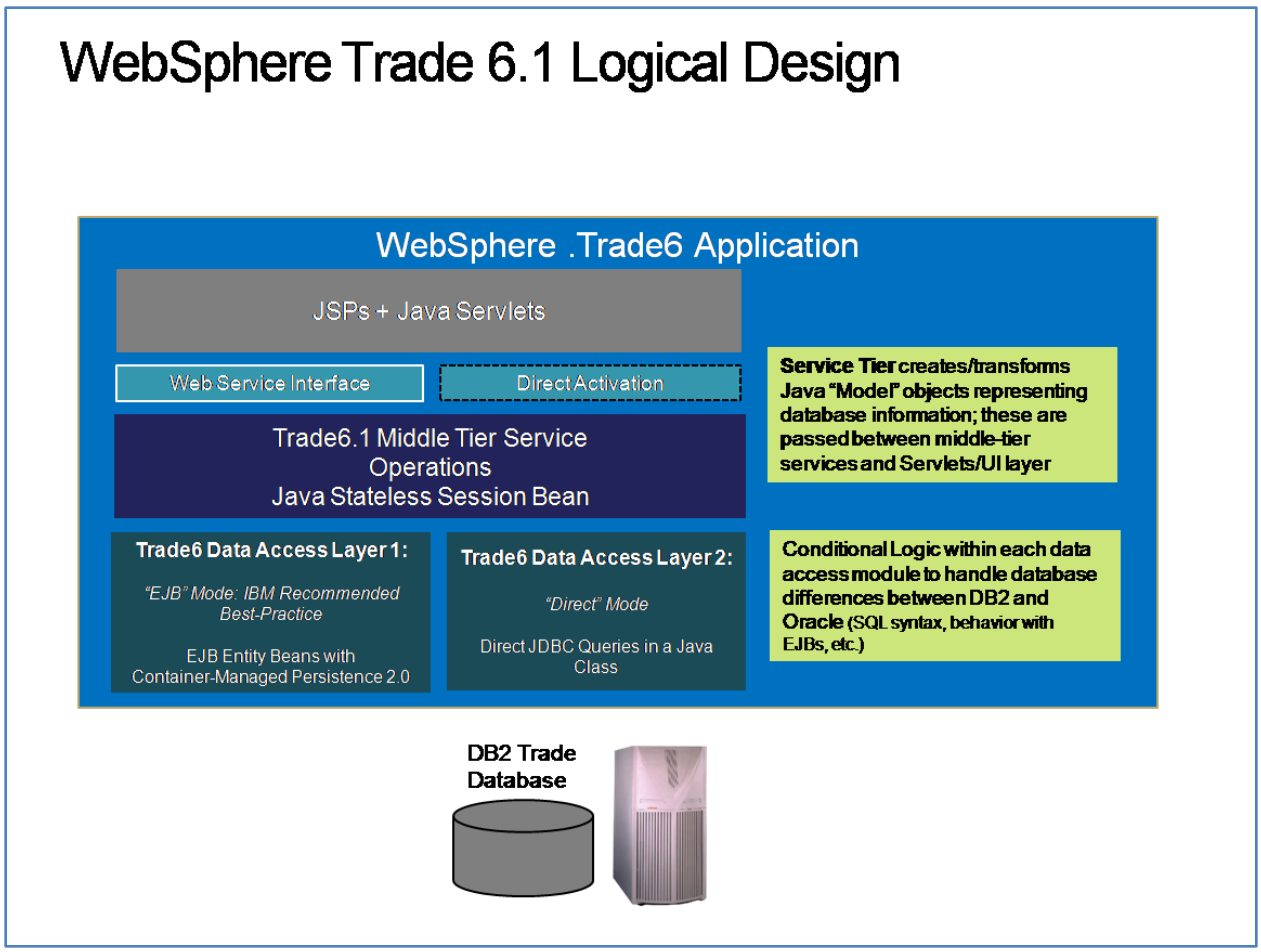

Figure 14: Trade 6.1 Logical Design

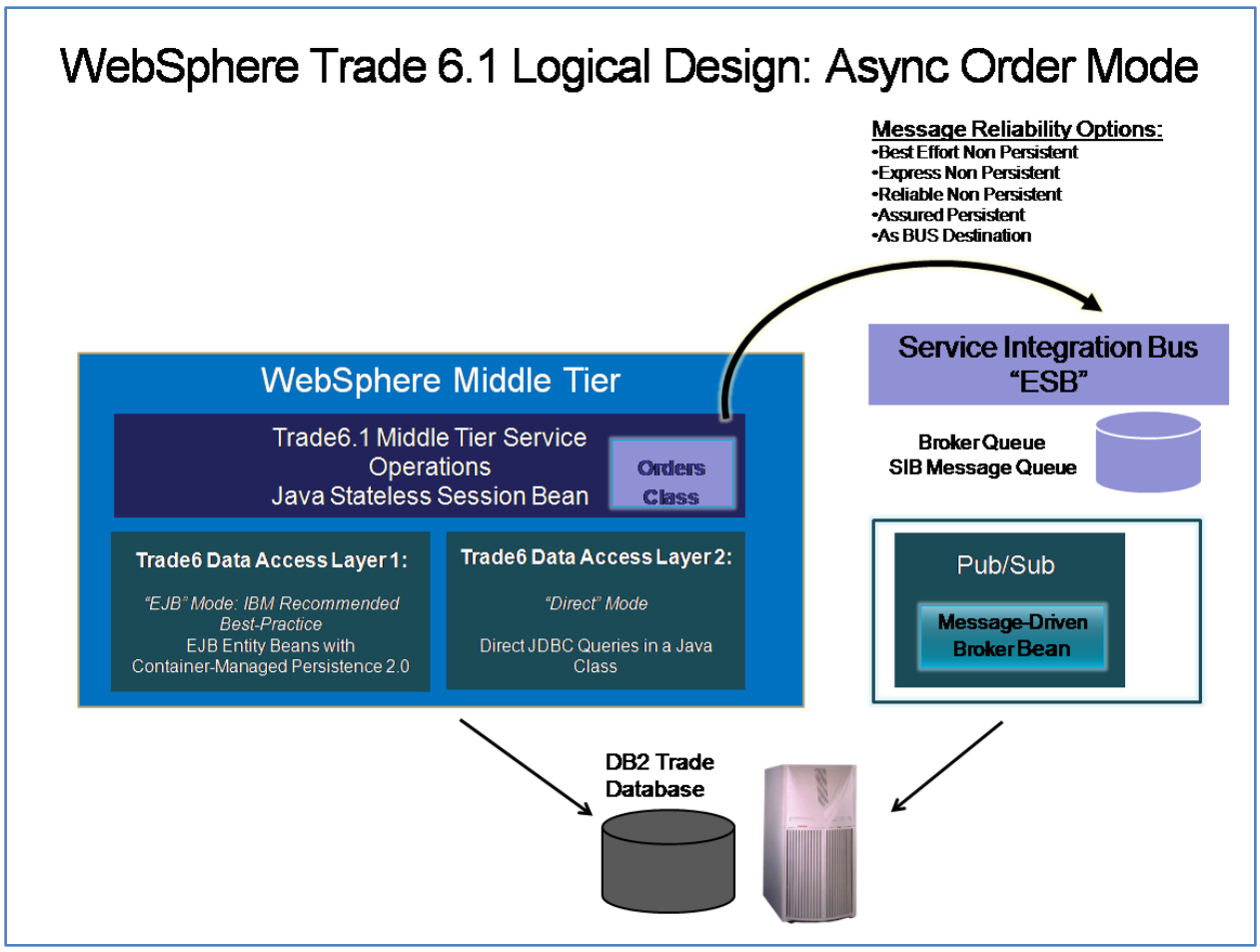

Figure 15: Asynchronous, Message-Oriented Processing of Orders

Conclusion

This paper presents an extensive array of benchmark comparisons between IBM WebSphere and .NET/Windows Server running an application server workload. The benchmark is based on the functional specification of IBM WebSphere Trade 6.1, as defined and developed by IBM for the WebSphere 6.1 platform. The .NET results are based on a migration of this application to .NET with the use of Windows Communication Foundation for the service layers. The .NET StockTrader is a best-practice performance implementation for the .NET platform, and is functionally and behaviorally equivalent to the tested Trade 6.1 application in the configurations tested. The benchmark results show the two platforms running in a variety of different configurations. With published source code for both implementations, we encourage customers to perform their own comparative testing; and also to use the .NET StockTrader application as a learning sample for various features of WCF and the Microsoft enterprise development technologies.

Appendix A: Pricing

The following pricing was used for the $/TPS calculations. Pricing is based on published list pricing for the products.

Pricing for the Web Service Tests

Pricing includes middle tier software licensing costs (OS + Application Server) for the primary application server/Web Service Host and the four remote Web Application servers used in the remote tests. Database software costs and middle tier/database hardware costs were not included. In the Web Services configuration, we priced WebSphere Express for the four Web Application Servers, and Network Deployment Edition for the Web Service Host. Network Deployment Edition is IBM's recommended enterprise application server, and includes its core enterprise features. Note that for the .NET/Microsoft Windows Server configuration, no separate application server is necessary: .NET is integrated into Windows Server and new versions are made available as free downloads on MSDN. There is also no redistribution license fee to redistribute the full .NET Framework runtime. Red Hat Advanced Platform was priced for the 4-CPU Linux application server tested, as this is required to support 4 CPUs. Windows Server 2003 R2 Enterprise was priced for the 4-CPU application server. Windows Server 2003 was run on the 4-Web Application Servers: Windows Standard pricing was calculated for these servers.

For the .NET configurations, the External Connector License, which allows unlimited Anonymous Web access (as used in the StockTrader Application) without CALs was added for all Windows Servers. Windows Standard Edition with Internet Connectors was priced for the 4 2-CPU Web Application Servers, Enterprise Edition for the 4-CPU primary Application Server/Web Service Host.

WebSphere Pricing Windows

Web Service Tests: 5 Systems

1 x WAS Network Deployment Edition:
($15,000 x 4 CPUs)
$60,000.00

1 x Windows Enterprise Edition:
$3,999.00

4 x WAS Express Edition:
($2,000 x 8 CPUs)
$16,000.00

4 x Windows Standard Edition (@ $1199.00 per copy)
$4,796.00

Total: $84,795.00

WebSphere Pricing Red Hat Linux

Web Service Tests: 5 Systems

1 x WAS Network Deployment Edition:
($15,000 x 4 CPUs)
$60,000.00

1 x RedHat Advanced Platform (V5)
$2,499.00

4 x WAS Express Edition:
($2,000 x 8 CPUs)
$16,000.00

4 x Windows Standard Edition (@ $1199.00 per copy)
$4,796.00

Total: $83,295.00

.NET Pricing (Windows Server 2003)

1 x Windows Enterprise Edition:
$3,999.00

4 x Windows Standard Edition (@ $1199.00 per copy)
$4,796.00

5 x External Connector License (@$1,999.00 per copy)
$9,995.00

Total: $18,790.00

Pricing for the Monolithic Application and Messaging Tests

1 Application Server System was used for the middle tier in these tests, with 4 CPUs.

WebSphere Pricing Windows

1 x WAS Network Deployment Edition:
($15,000 x 4 CPUs)
$60,000.00

1 x Windows Enterprise Edition:
$3,999.00

1 x External Connector License
$1,999.00

Total: $65,998.00

WebSphere Pricing Linux

1 x WAS Network Deployment Edition:
($15,000 x 4 CPUs)
$60,000.00

1 x RedHat Advanced Platform (V5)
$2,499.00

Total: $62,499.00

.NET Pricing (Windows Server 2003)

1 x Windows Enterprise Edition:
$3,999.00

1 x External Connector License (@$1,999.00 per copy)
$1,999.00

Total: $5,998.00

Appendix B: Performance Monitor Captures

Notes: The 4 x 2-Processor Web Server machines are hyper-threaded, so show 4 CPUs in Performance Monitor. #Bytes All Heaps and Requests Queued are .NET Performance Counters, and will show as zero during WebSphere benchmark runs.

What are We Looking For?

1. Near 100% CPU saturation of the Web Application Server Under Test (this is always the 4 x 1.8 GHz AMD Opteron System).
2. Response times (the right hand Mercury Console Window) less than .5 seconds---request queuing is just beginning, but we are not over-stressing the computer; we want to reach 100% CPU saturation or as close as possible to get to peak throughput, but anything beyond would cause throughput to fall as systems begin to queue requests.
3. Database CPU loads well under 100%. Note that you will see *higher* database CPU loads for benchmark configuration runs that are pushing more TPS through the system. The more TPS the middle tier is able to handle, the more database requests it is making per second.
4. Well under 100% CPU load for the 4 Web Application servers used in the remote Web Service tests; plus roughly equal CPU loads for each run across these four Web App client boxes.
5. Note for Web Service Benchmark runs that are pushing more TPS through the system, more user load is required across the four Web Application Servers; hence, you will see *higher* CPU load because these boxes are handling more concurrent users and more TPS to saturate the Web Service Host. For example, the netTcp WCF benchmark requires much greater loads to saturate the server, produces higher TPS; hence the database and the four ASP.NET clients are also handling more load to produce this TPS rate, as expected. In summary, always judge a CPU utilization rate based on the TPS it is handling at that CPU utilization.
6. Note these shots were taken largely during warm up runs, after steady state had been achieved. Actual results are based on 30 minute measurement intervals in the official benchmark runs, averaged by Mercury Analysis tools after completion and not read from the real-time display console shown here.

WebSphere 6.1 Windows

Web Services Benchmark: EJB Mode

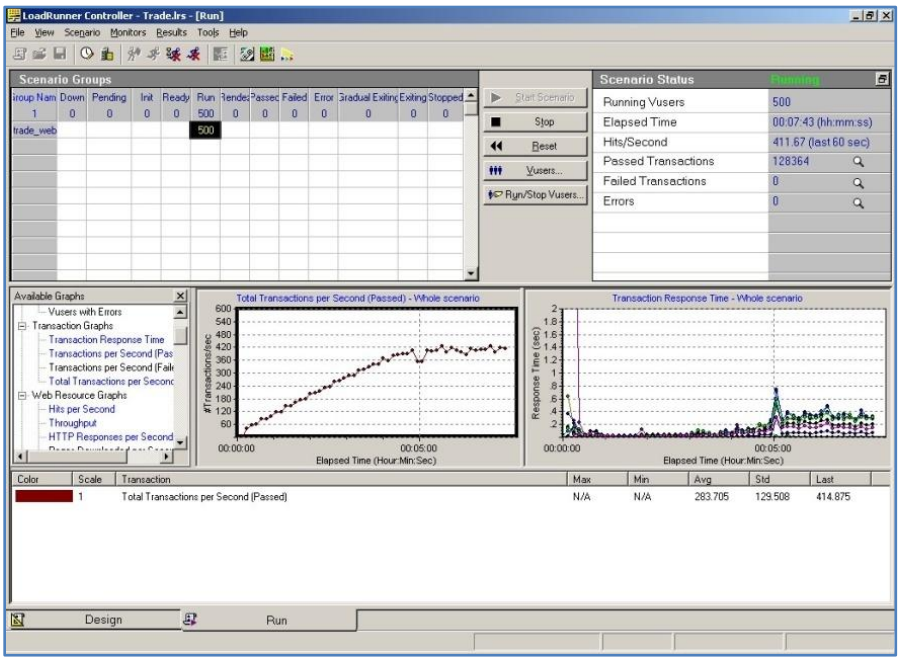

Figure 16: Mercury LoadRunner

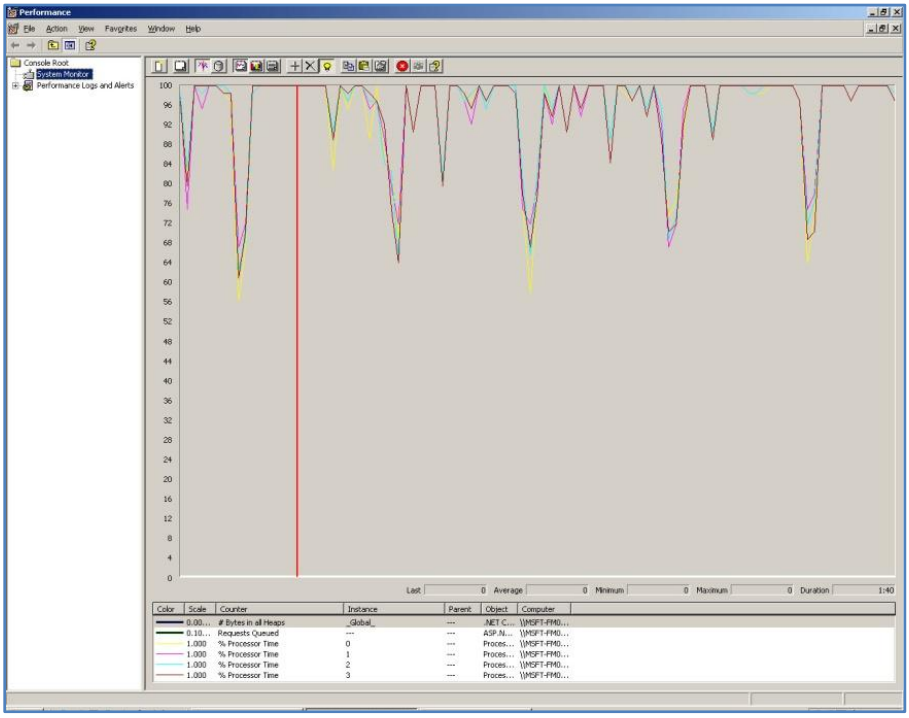

Figure 17: WebSphere Web Service Host

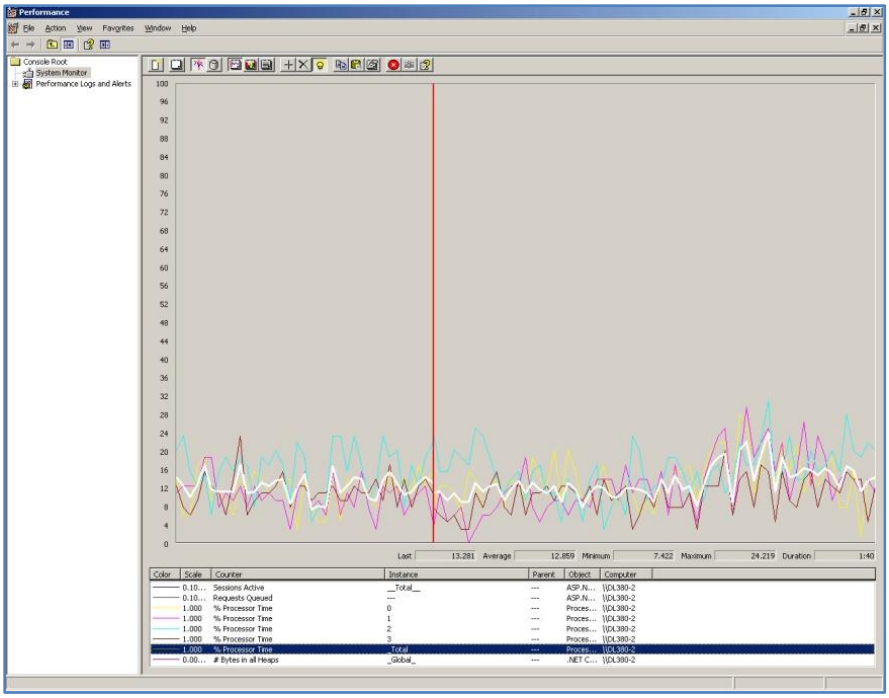

Figure 18: JSP App Server1

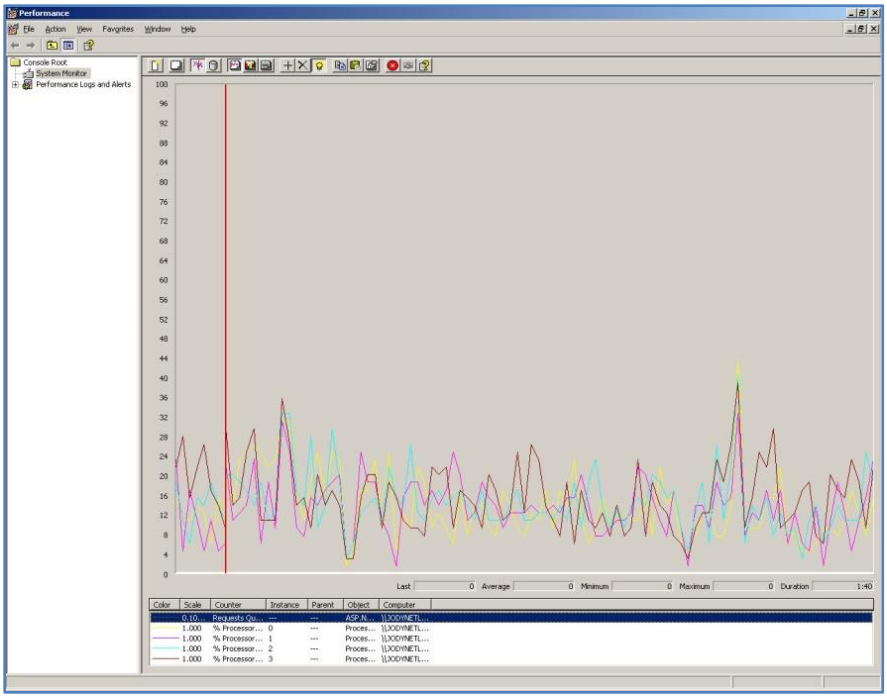

Figure 19: JSP App Server2

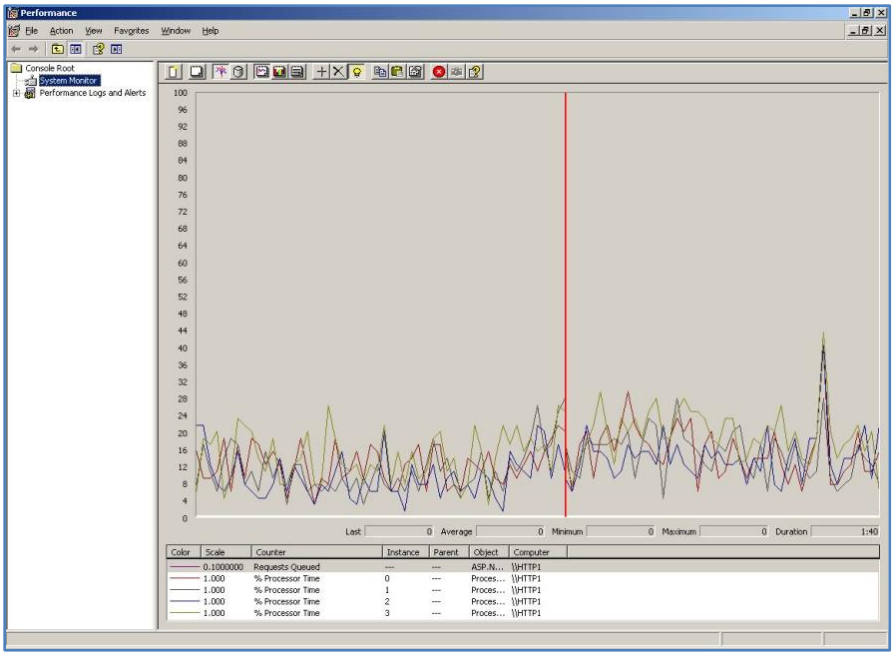

Figure 20: JSP App Server 3

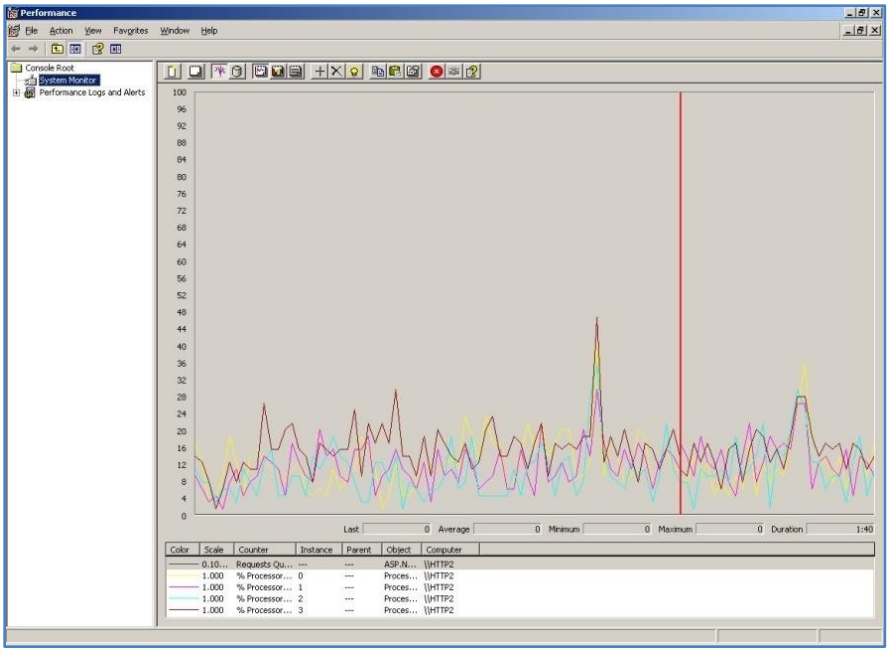

Figure 21: JSP App Server 4

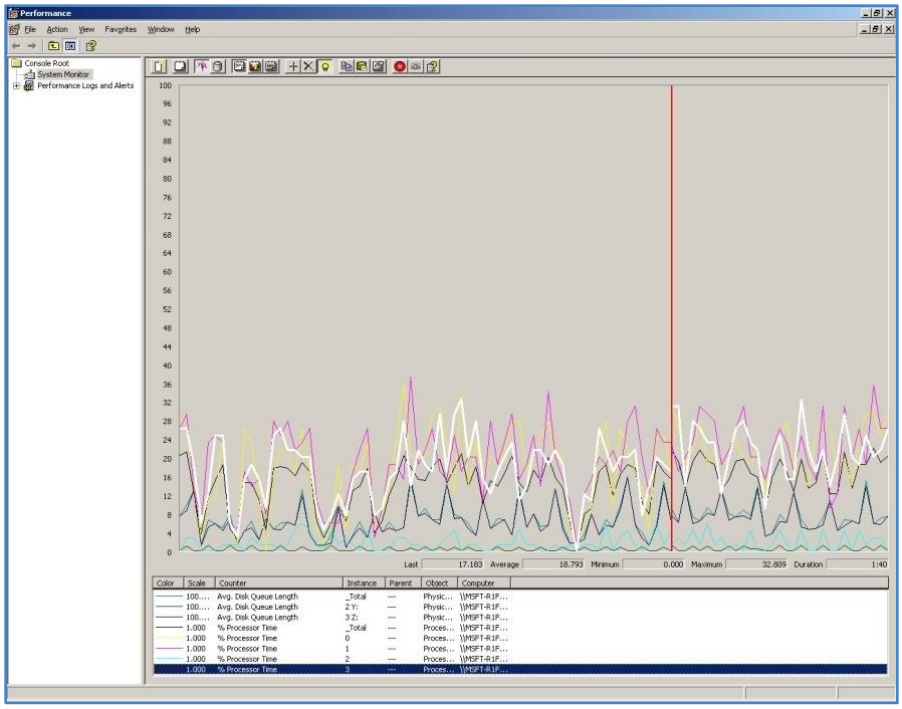

Figure 22: DB2

Web Services Benchmark: Direct (JDBC) Mode

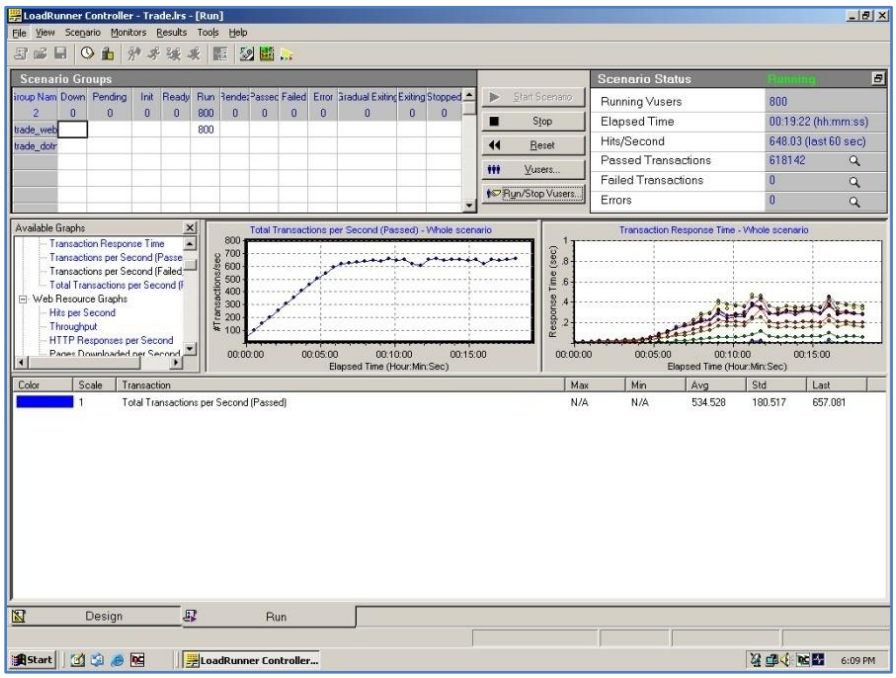

Figure 23: Mercury LoadRunner

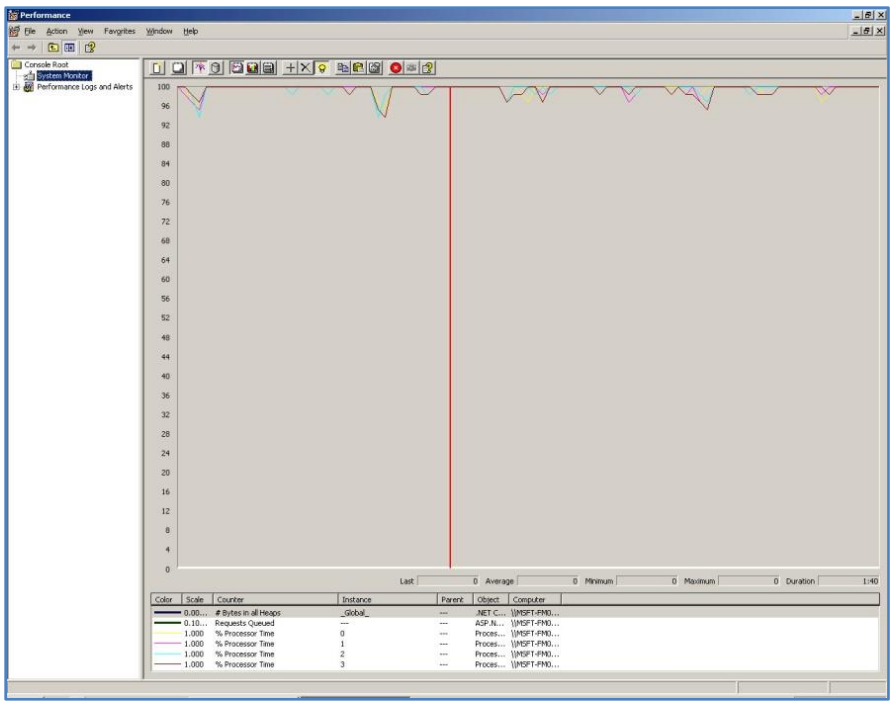

Figure 24: WebSphere Web Service Host

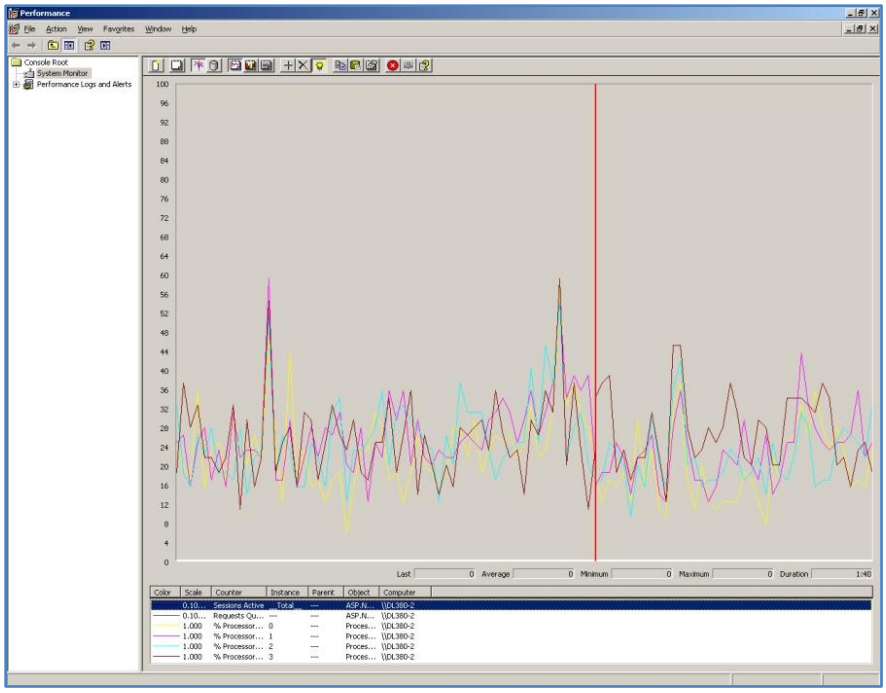

Figure 25: JSP App Server1

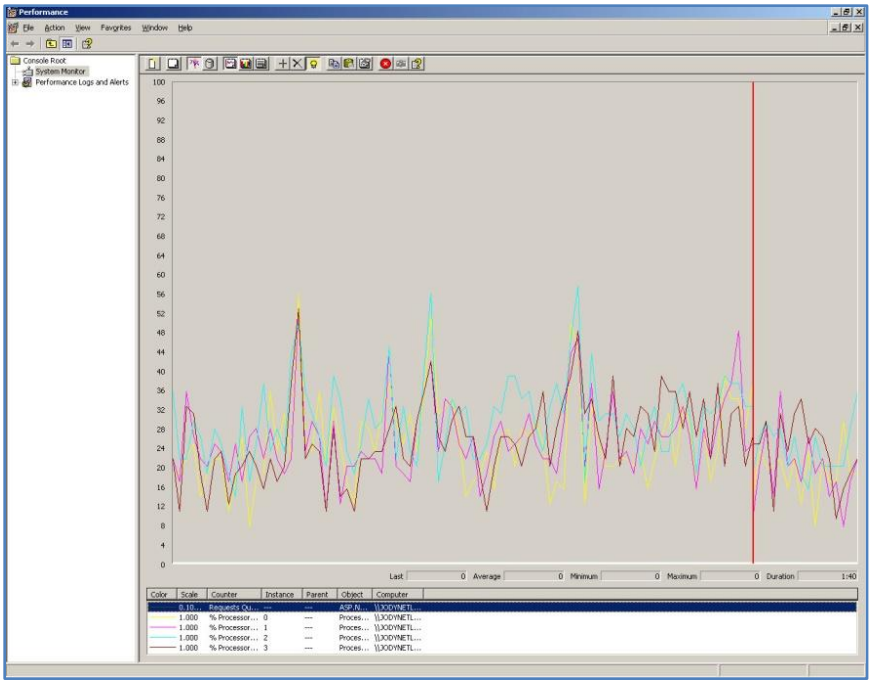

Figure 26: JSP App Server2

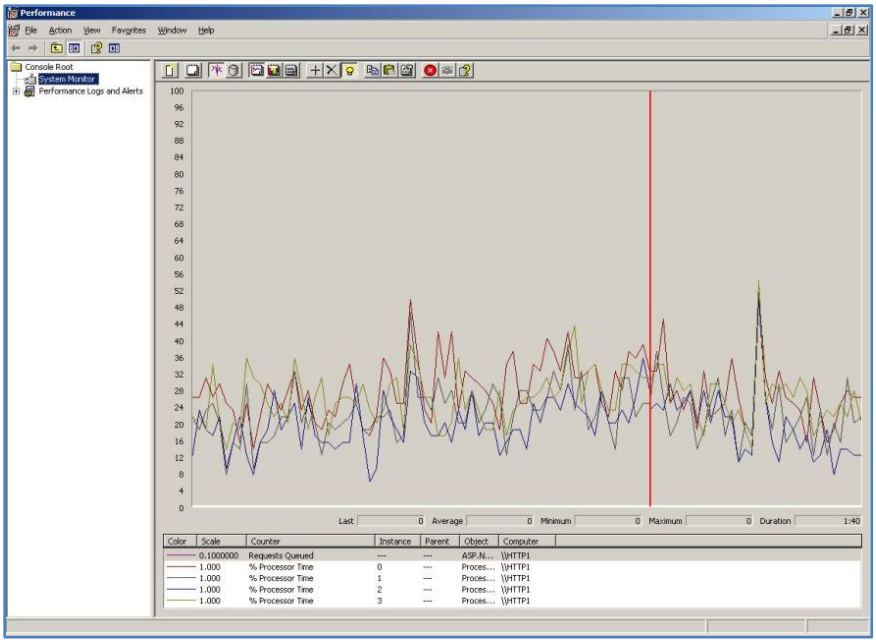

Figure 27: JSP App Server 3

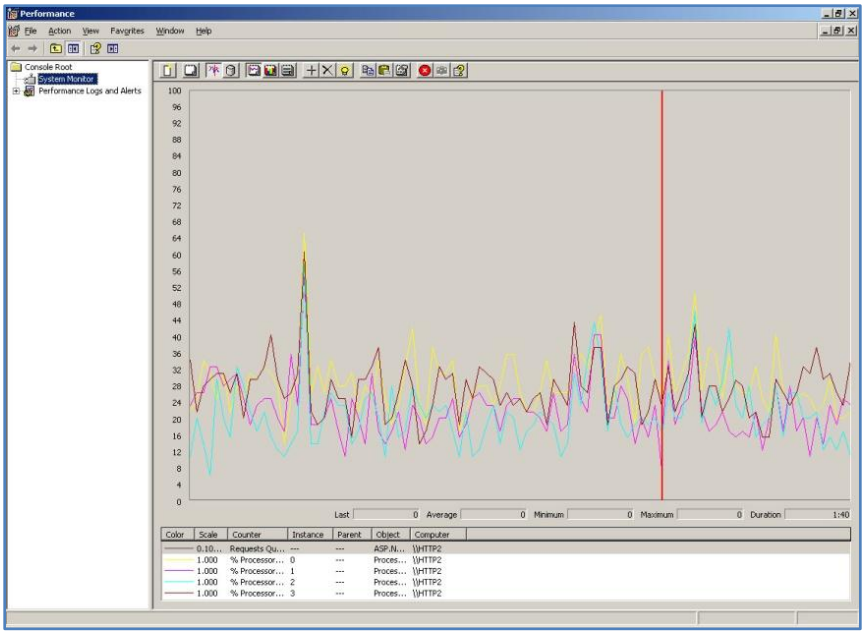

Figure 28: JSP App Server 4

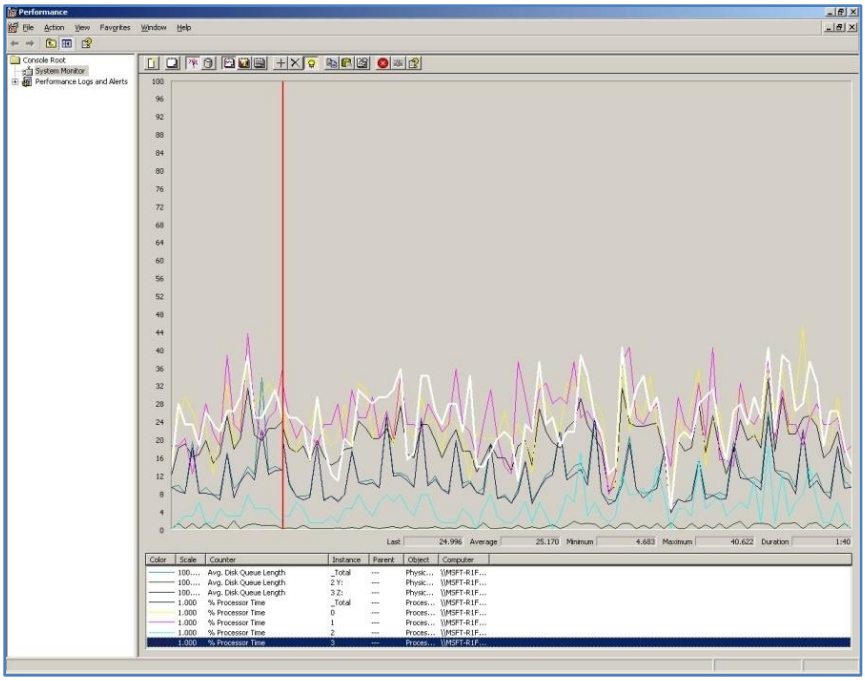

Figure 29: DB2

Messaging Benchmark Persistent Queue TwoPhase– EJB Mode

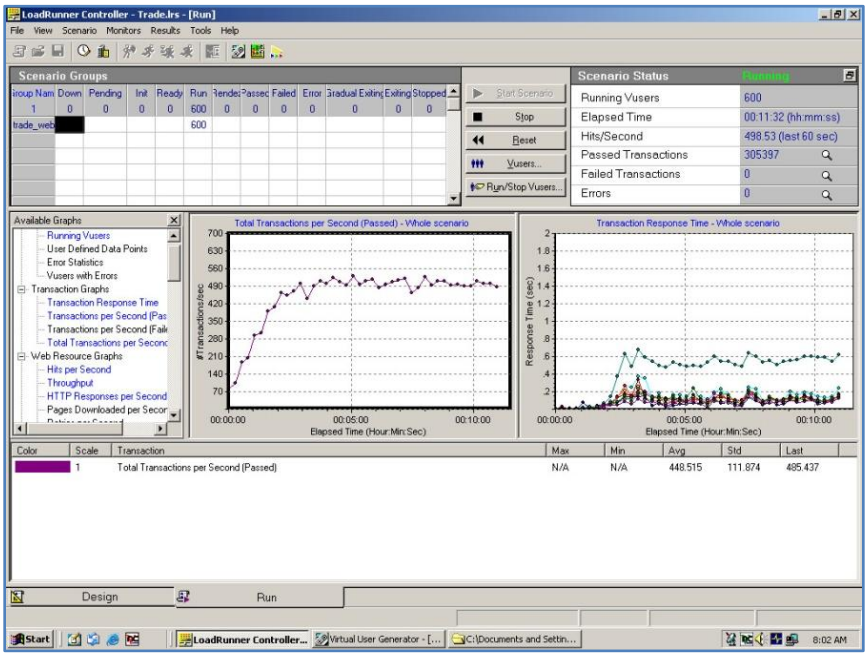

Figure 30: Mercury LoadRunner

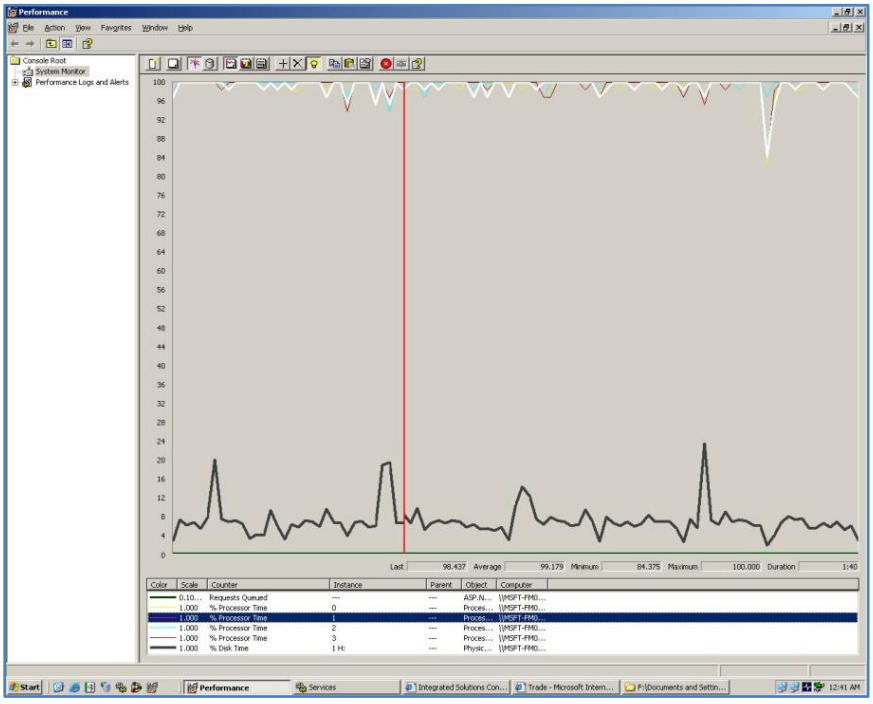

Figure 31: Application Server

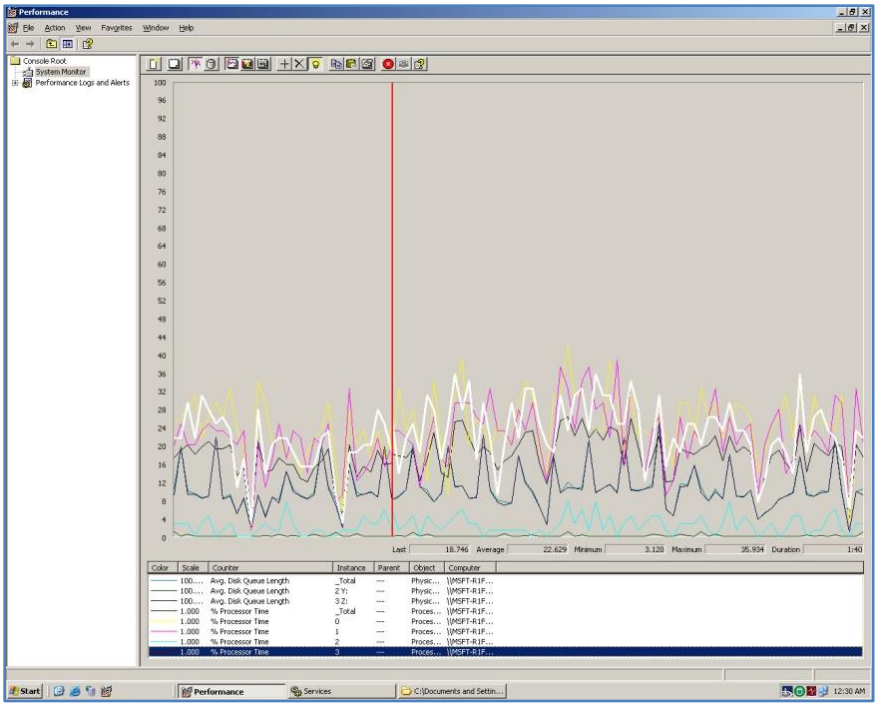

Figure 32: DB2

Messaging Benchmark Persistent Queue TwoPhase– Direct/JDBC Mode

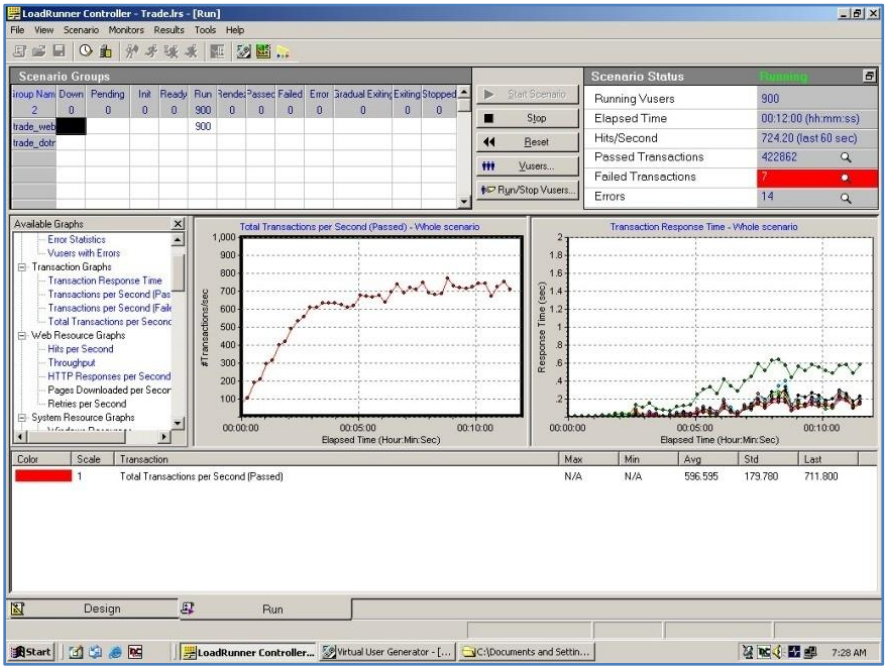

Figure 33: Mercury LoadRunner

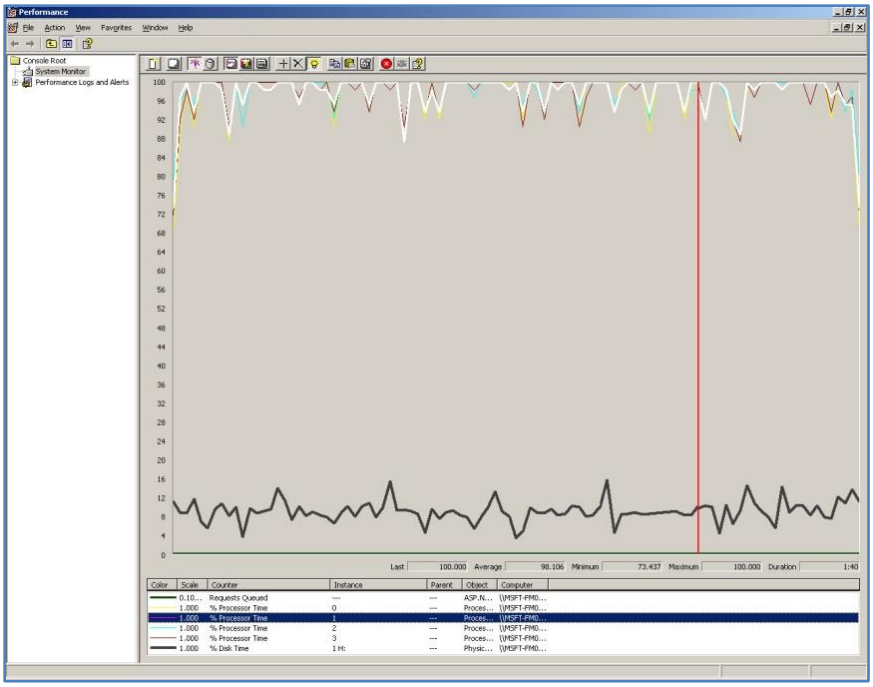

Figure 34: Application Server

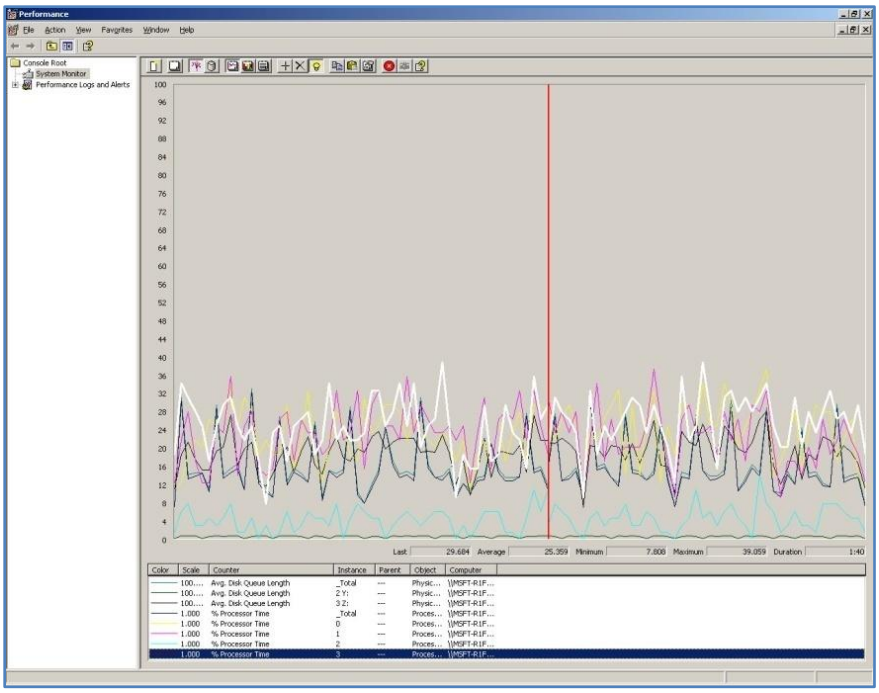

Figure 35: DB2

Messaging Benchmark NonPersistent Queue OnePhase – EJB Mode

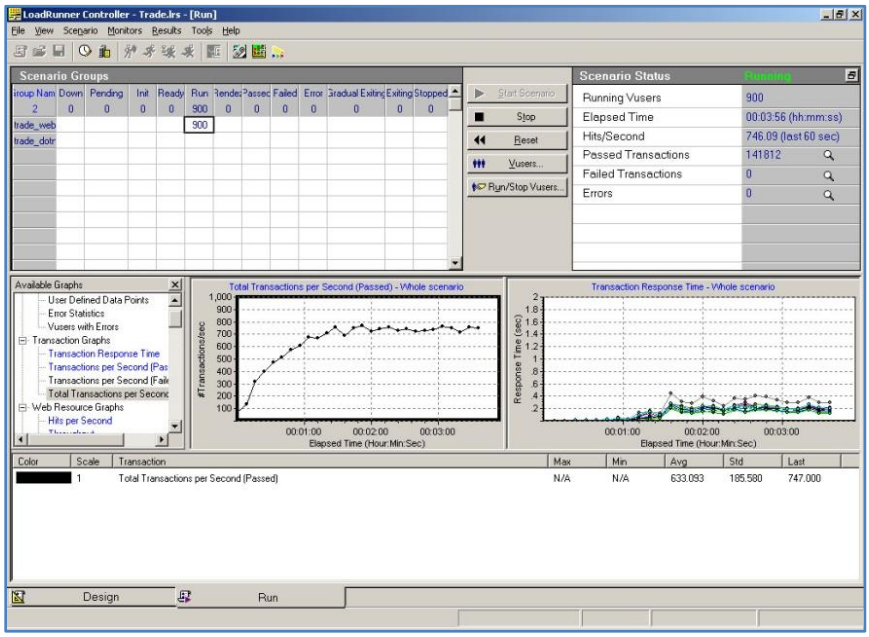

Figure 36: Mercury LoadRunner

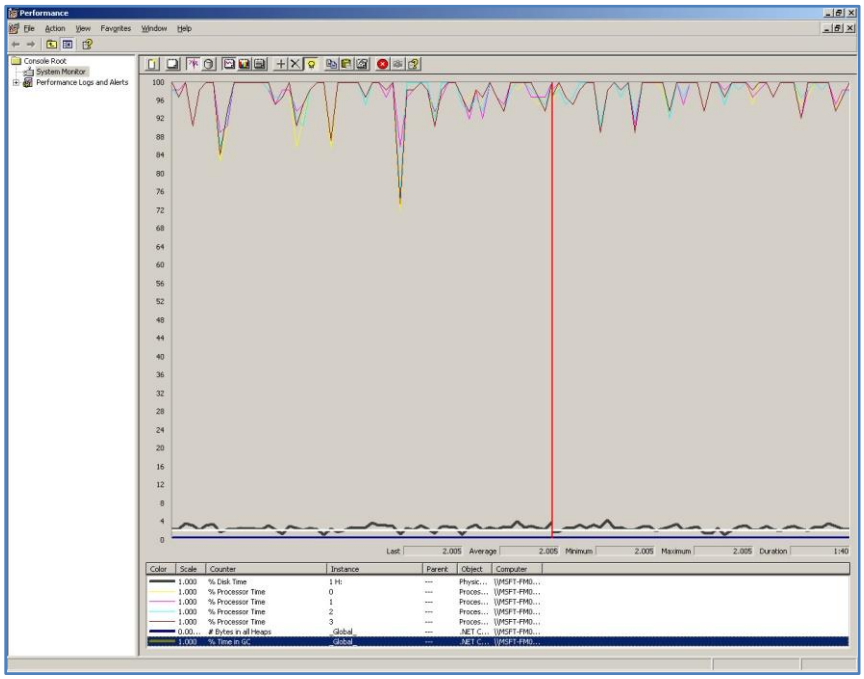

Figure 37: Application Server

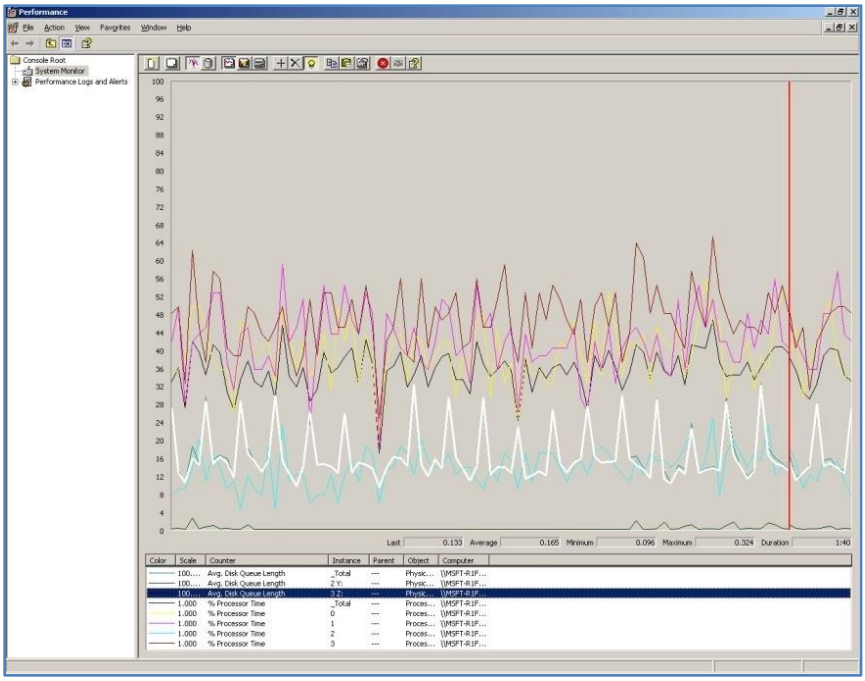

Figure 38: DB2

Messaging Benchmark NonPersistent Queue OnePhase Direct/JDBC Mode

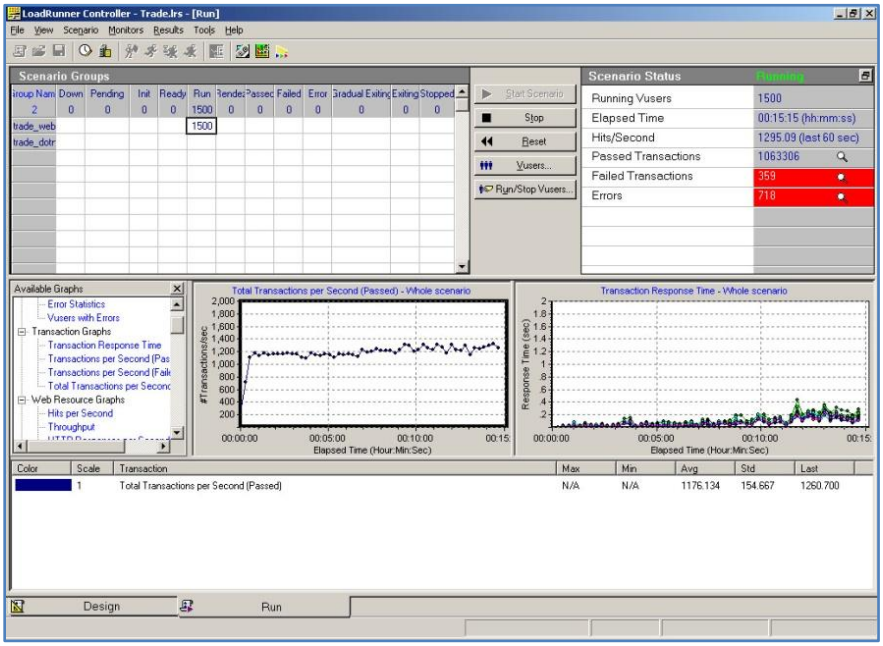

Figure 39: Mercury LoadRunner

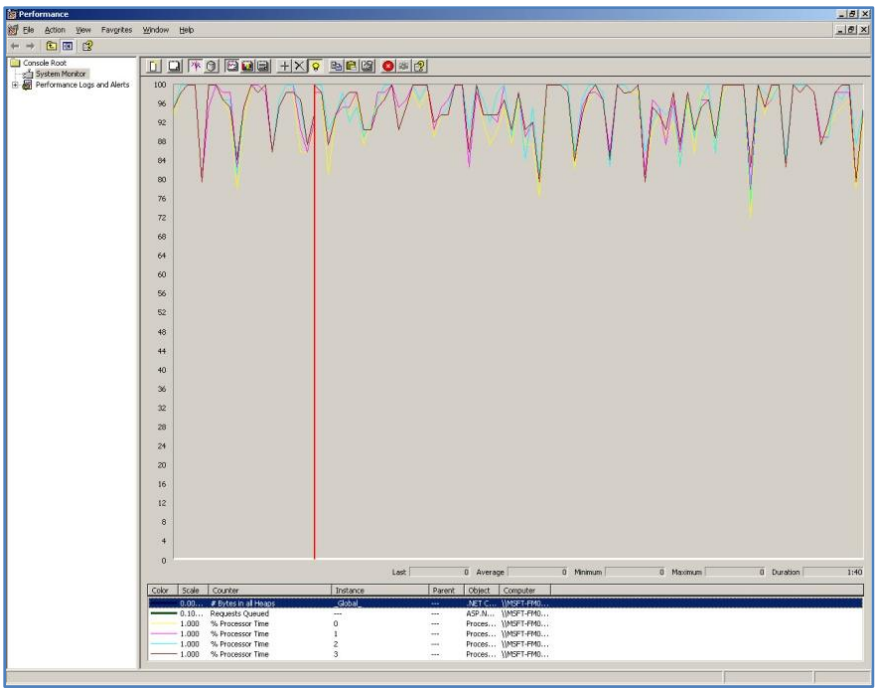

Figure 40: Application Server

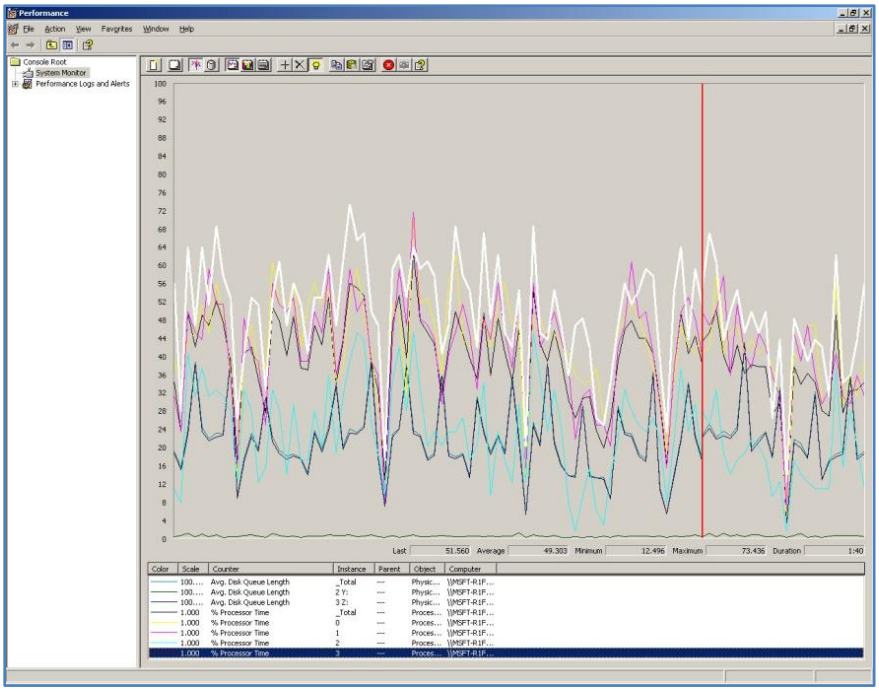

Figure 41: DB2

Monolithic Application: Synchronous Orders and Standard (non-remoted) Business Tier/Web Tier Access

EJB Mode

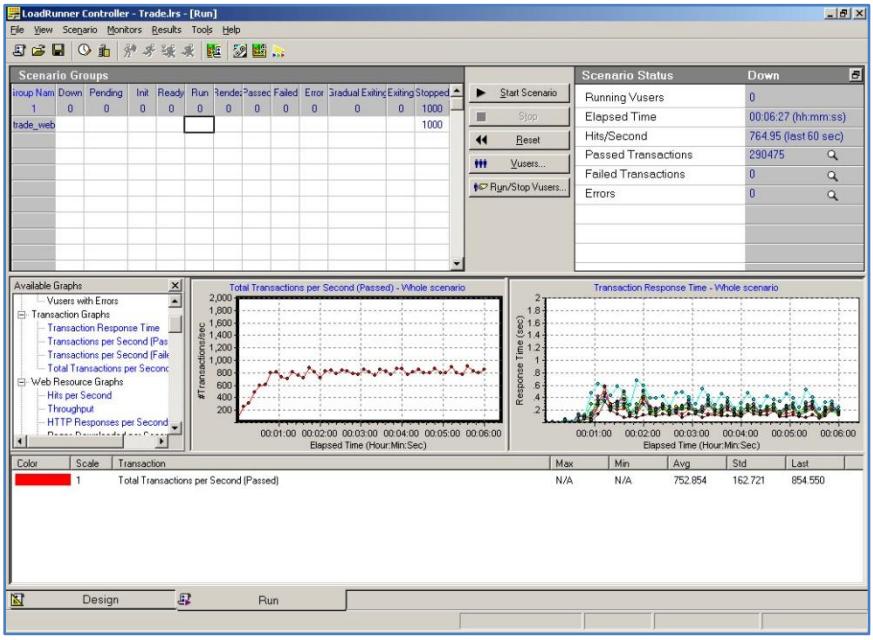

Figure 42: Mercury LoadRunner

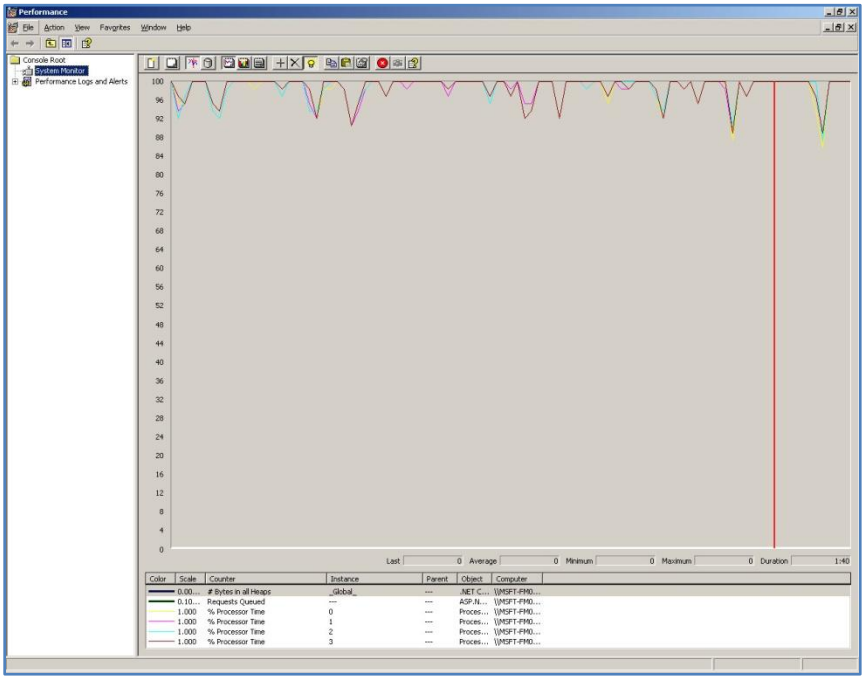

Figure 43: Application Server

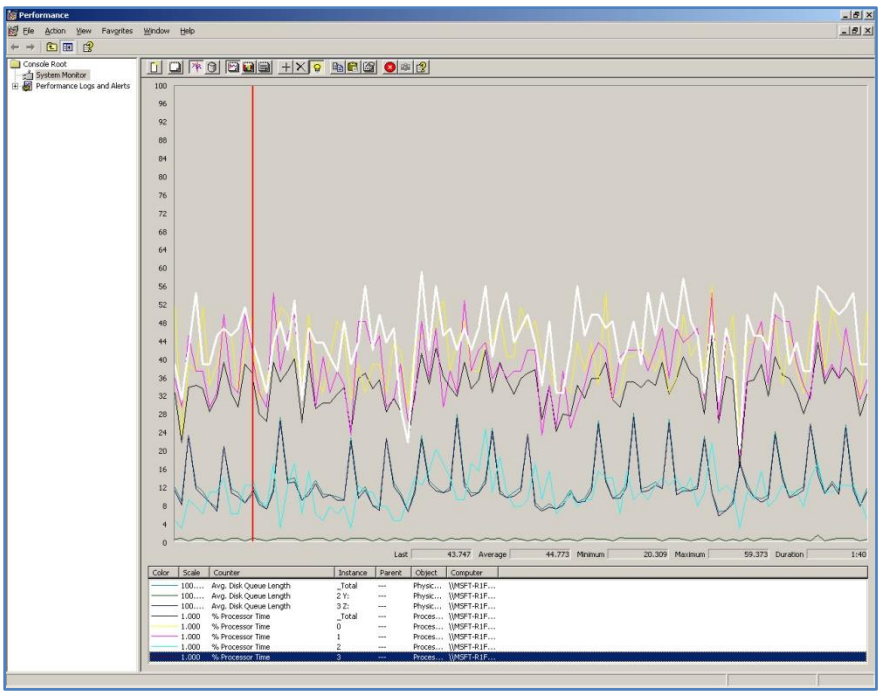

Figure 44: DB2

Direct Mode

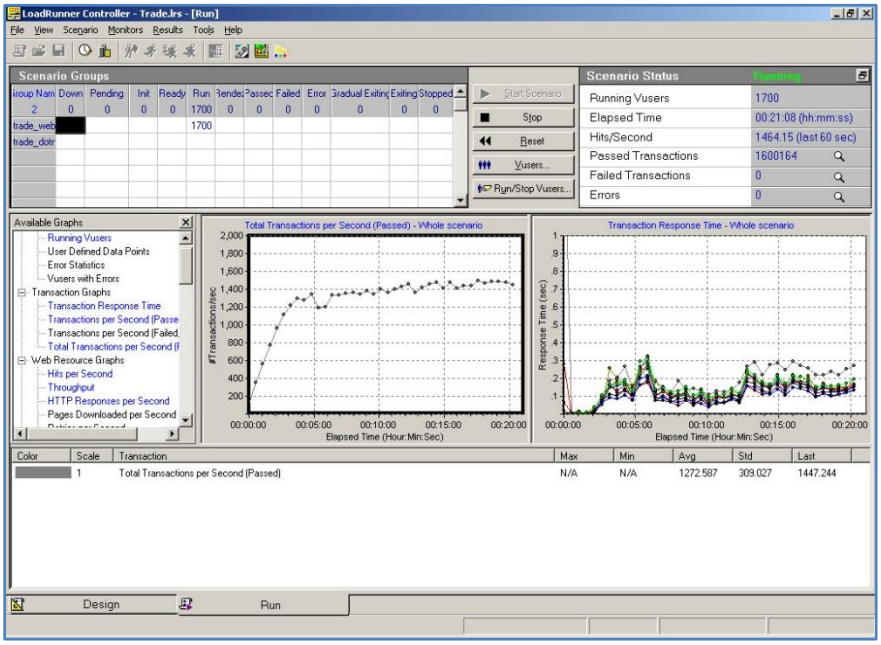

Figure 45: Mercury LoadRunner

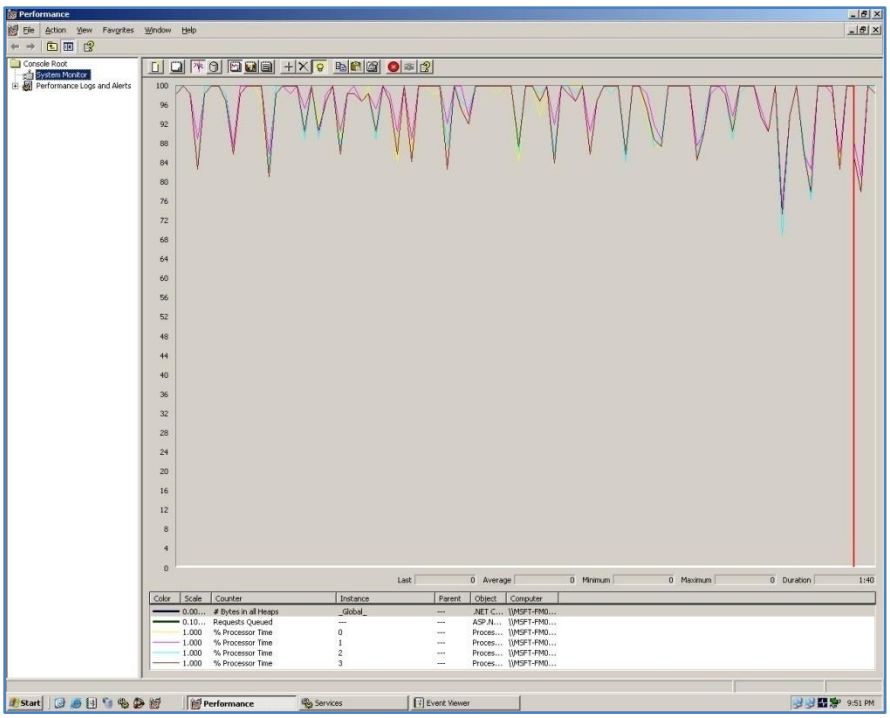

Figure 46: Application Server

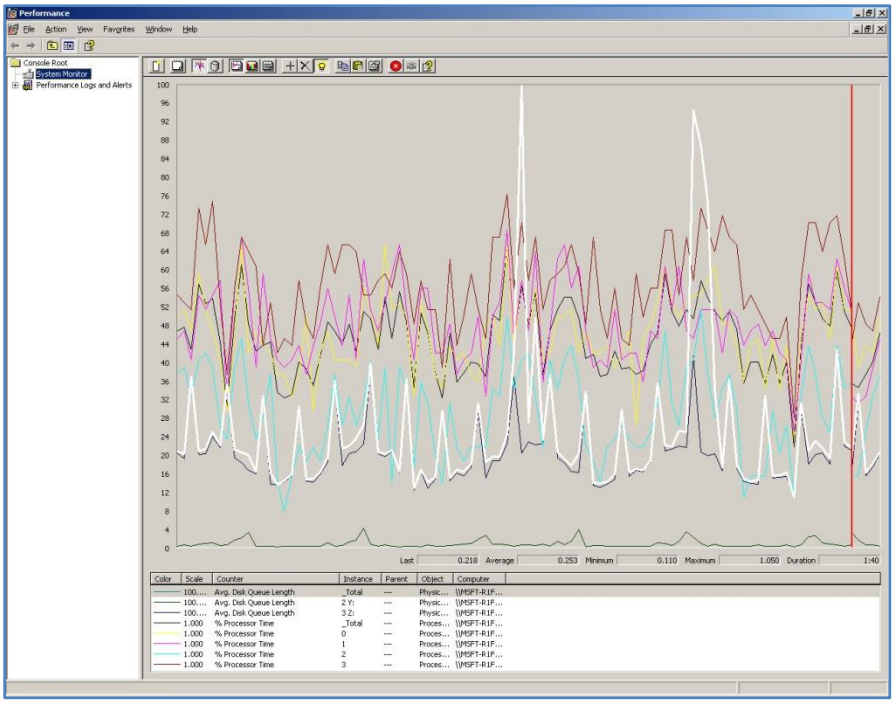

Figure 47: DB2

WebSphere 6.1 Linux

Web Services Benchmark: EJB Mode

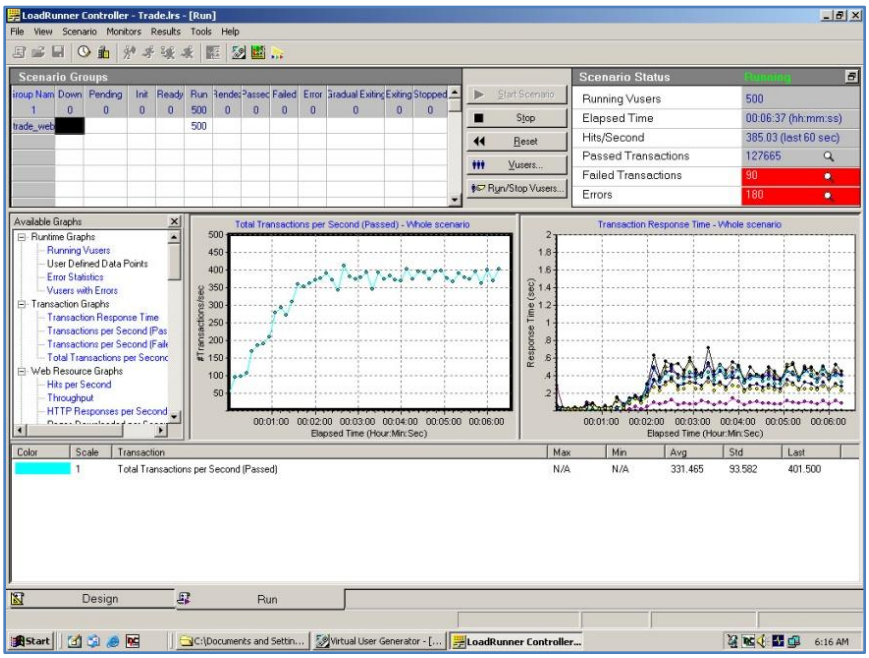

Figure 48: Mercury LoadRunner

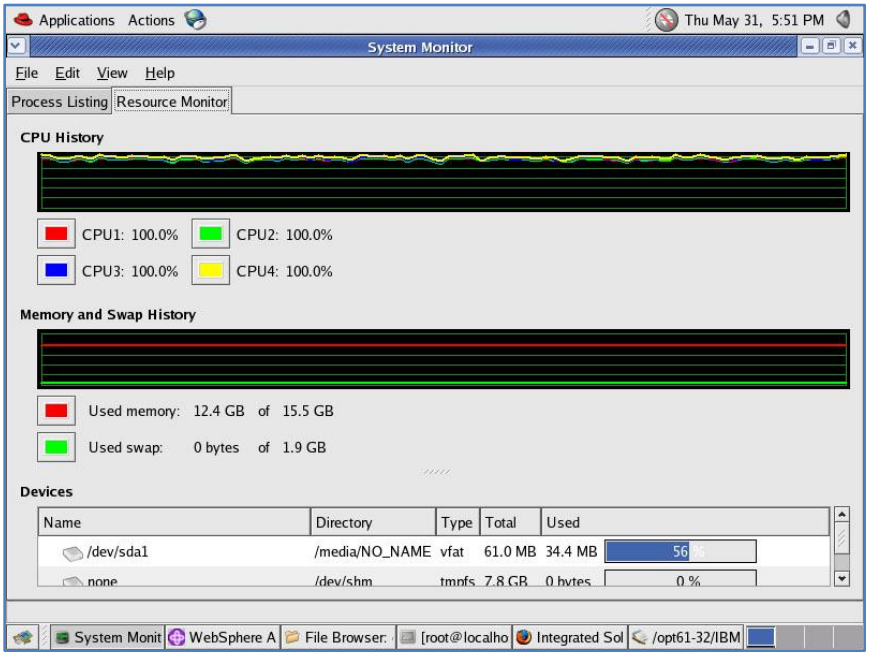

Figure 49: Web Service Host

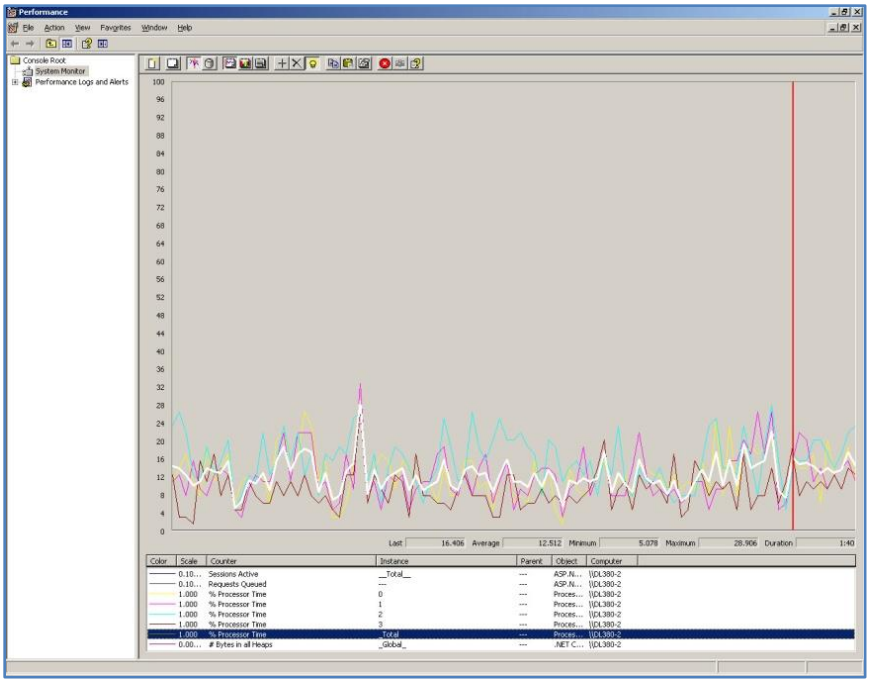

Figure 50: JSP App Server1

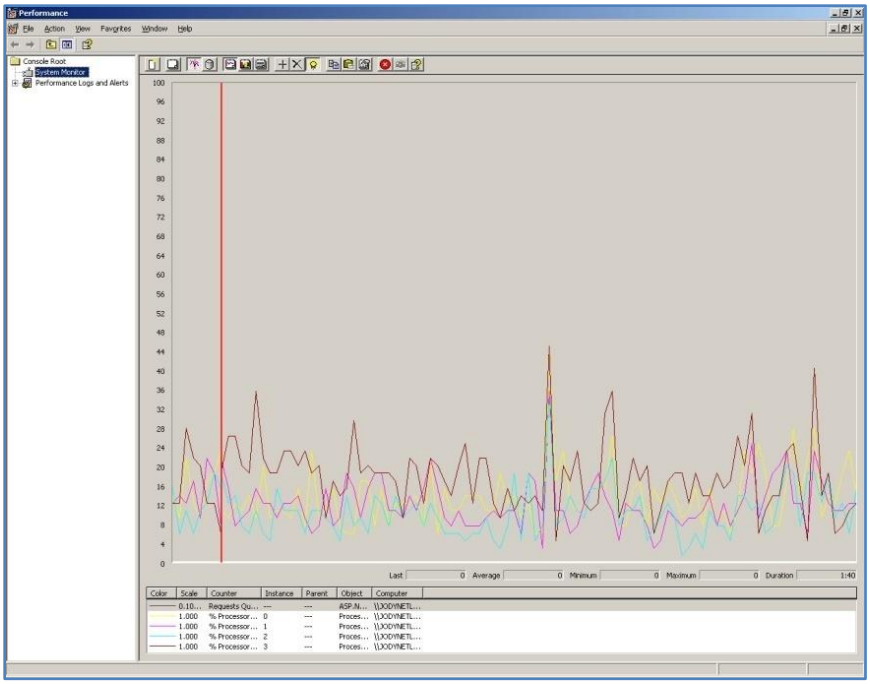

Figure 51: JSP App Server2

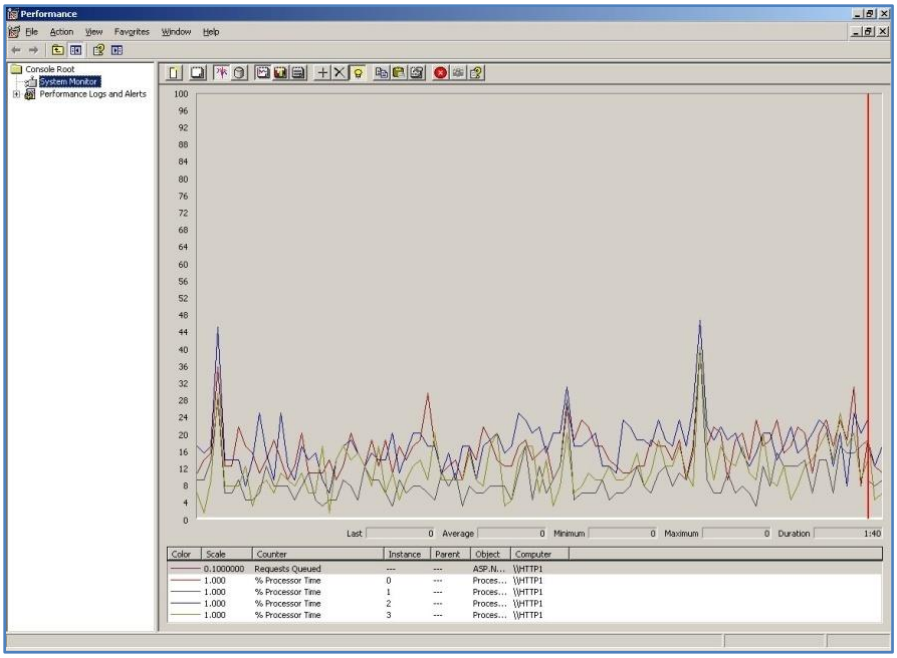

Figure 52: JSP App Server 3

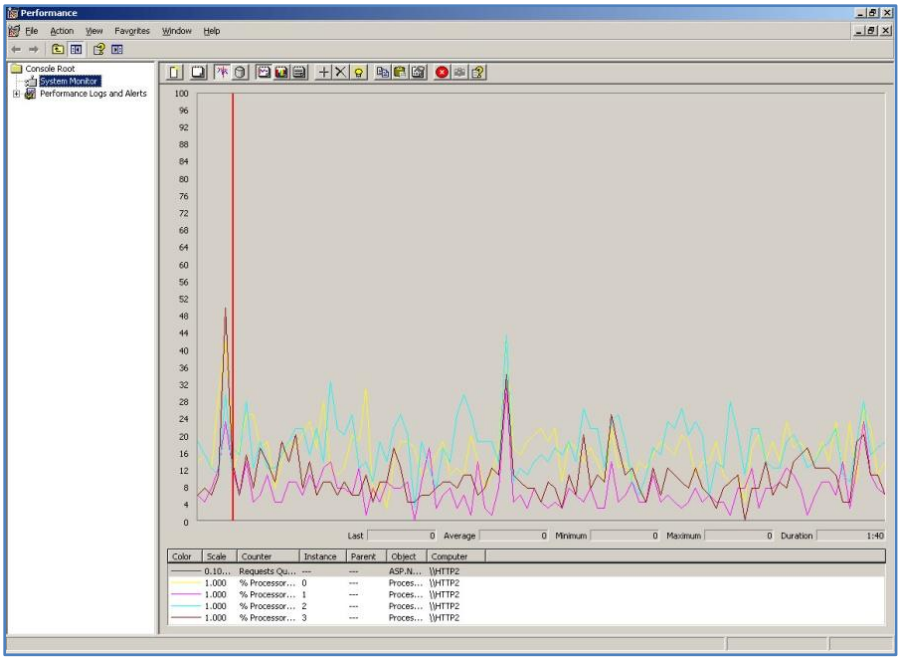

Figure 53: JSP App Server 4

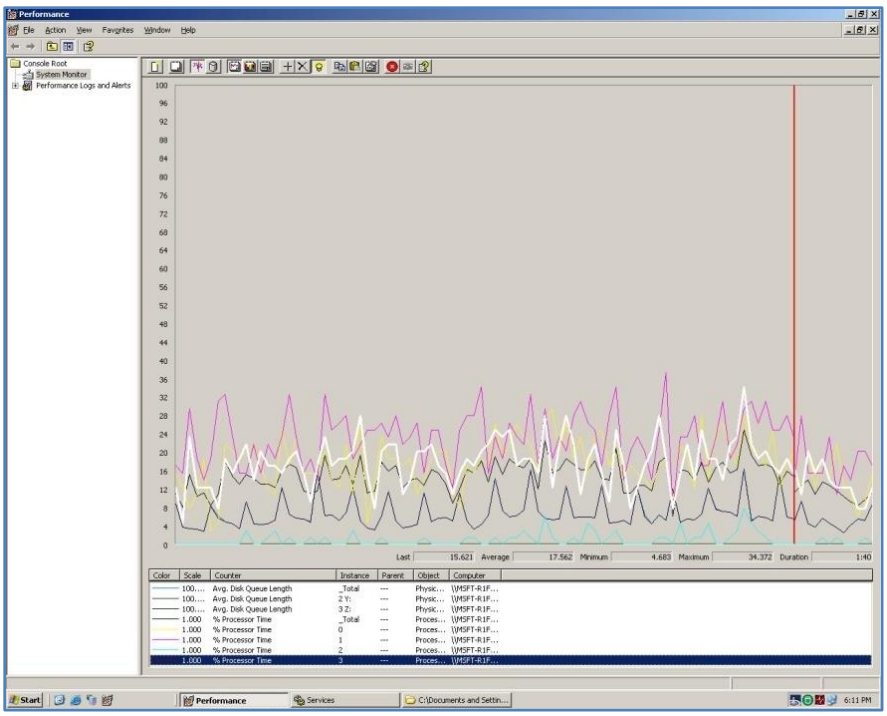

Figure 54: DB2

Web Services Benchmark: Direct (JDBC) Mode

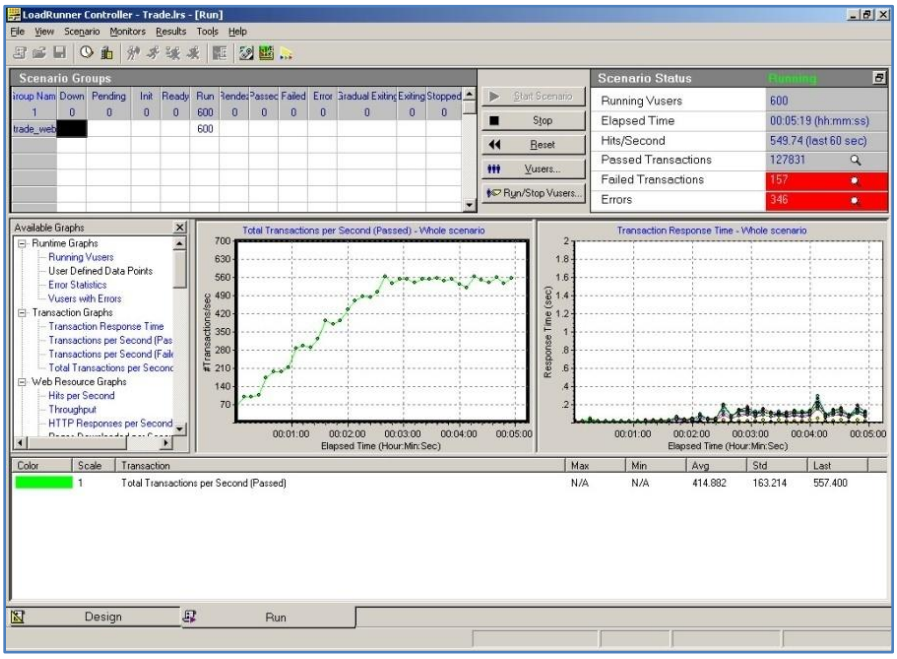

Figure 55: Mercury LoadRunner

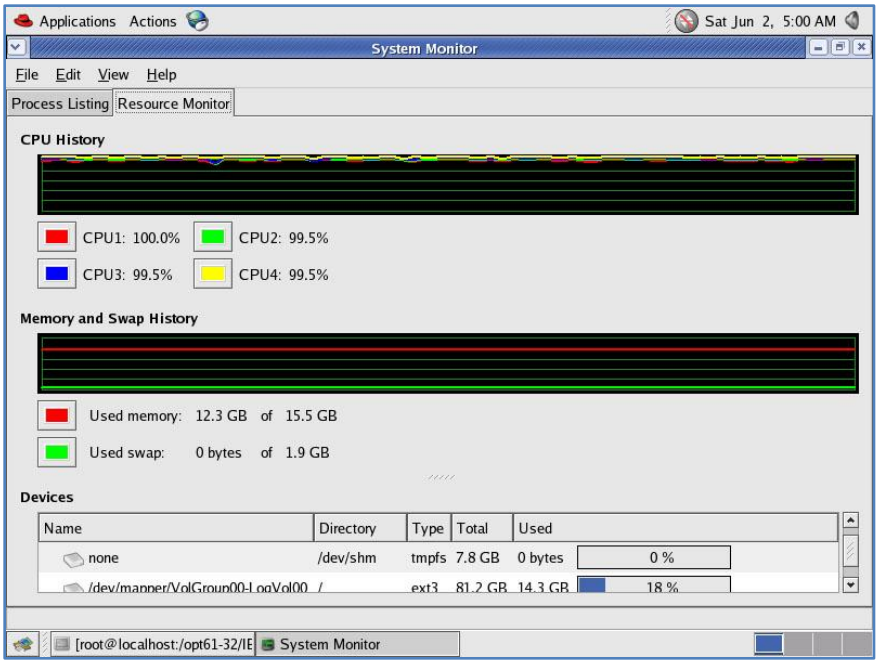

Figure 56: Web Service Host

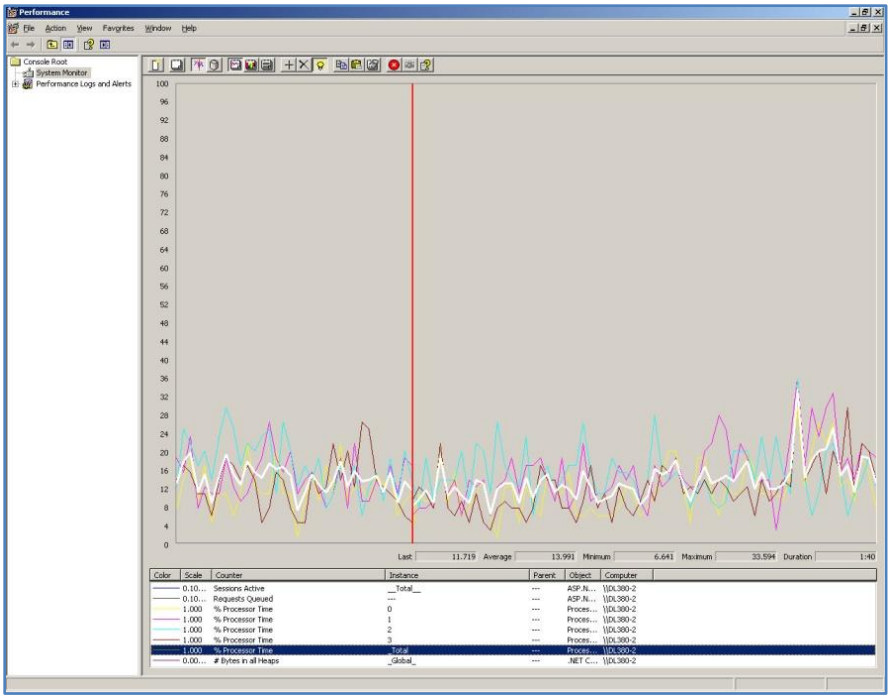

Figure 57: JSP App Server1

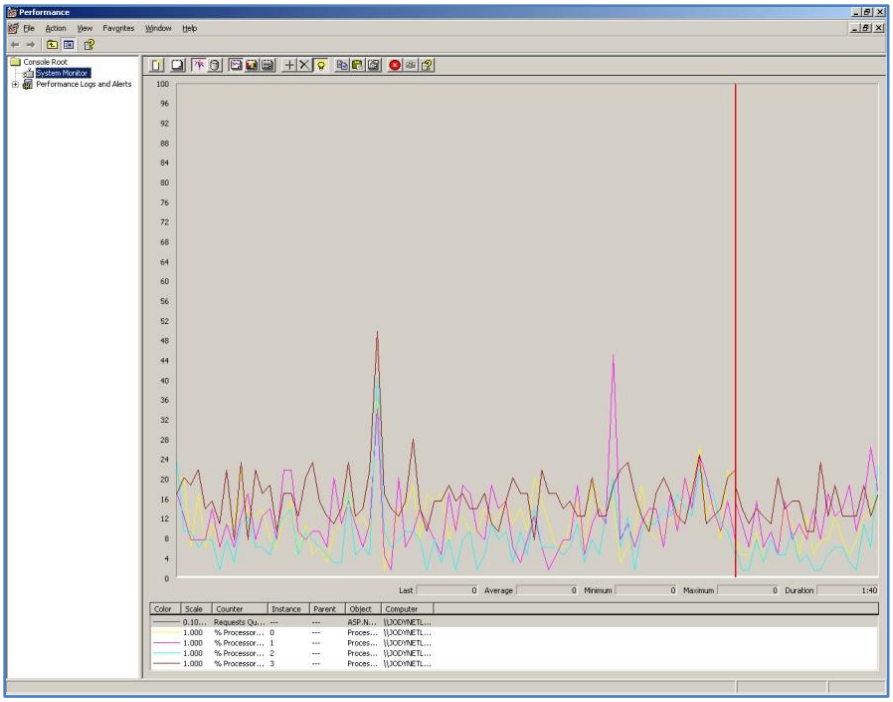

Figure 58: JSP App Server2

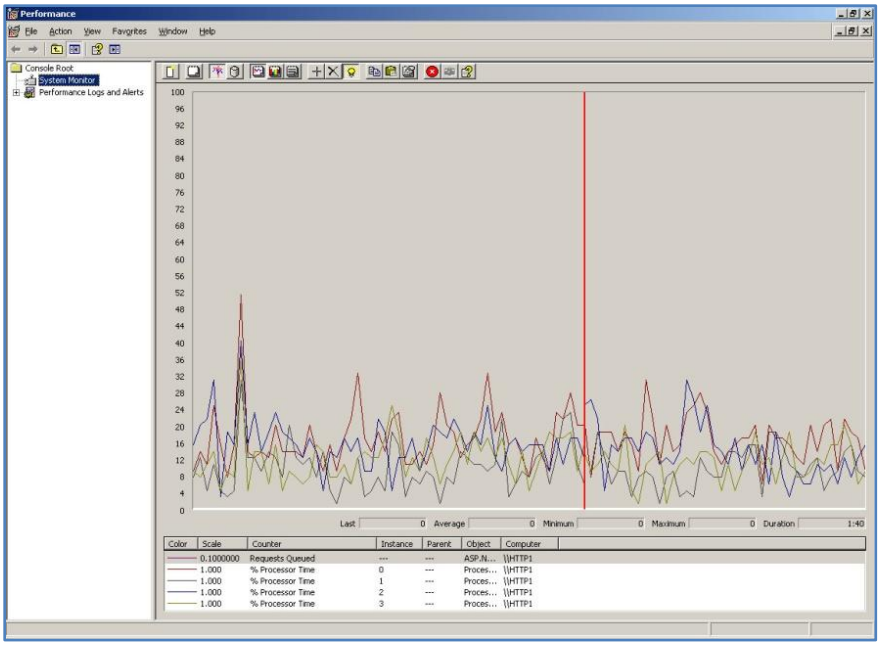

Figure 59: JSP App Server 3

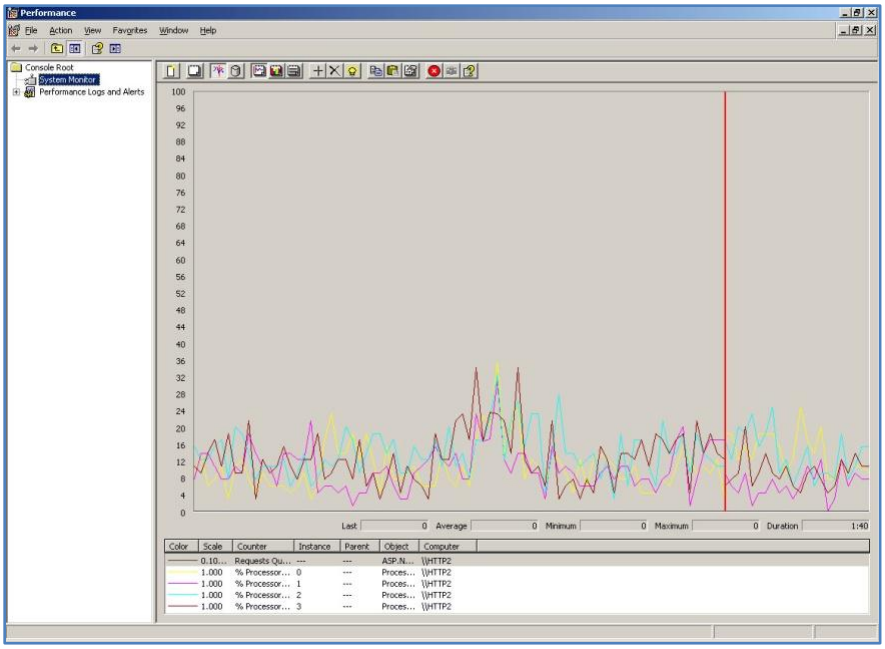

Figure 60: JSP App Server 4

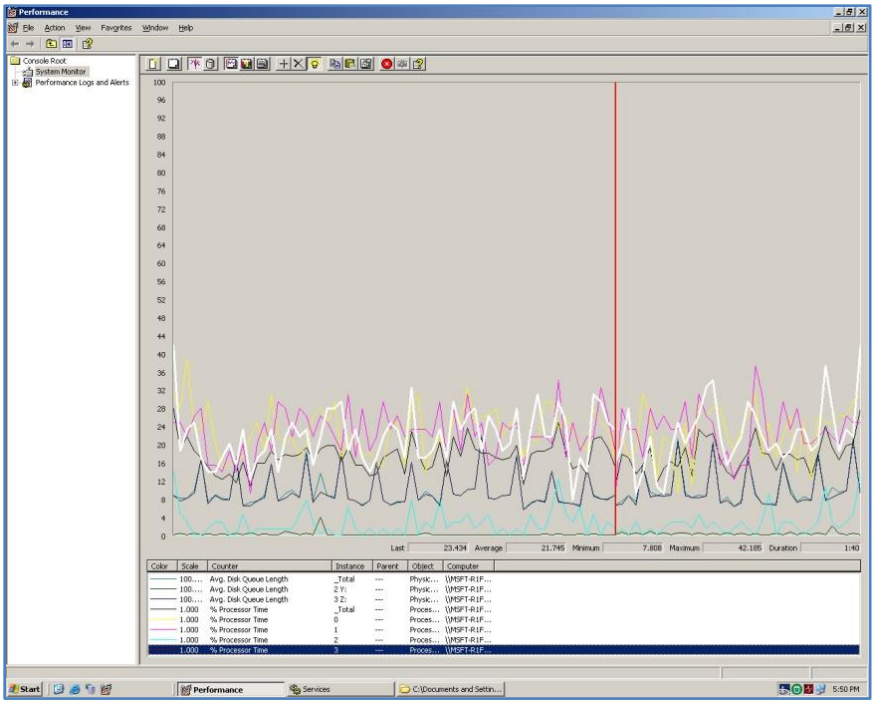

Figure 61: DB2

Messaging Benchmark Persistent Queue TwoPhase– EJB Mode

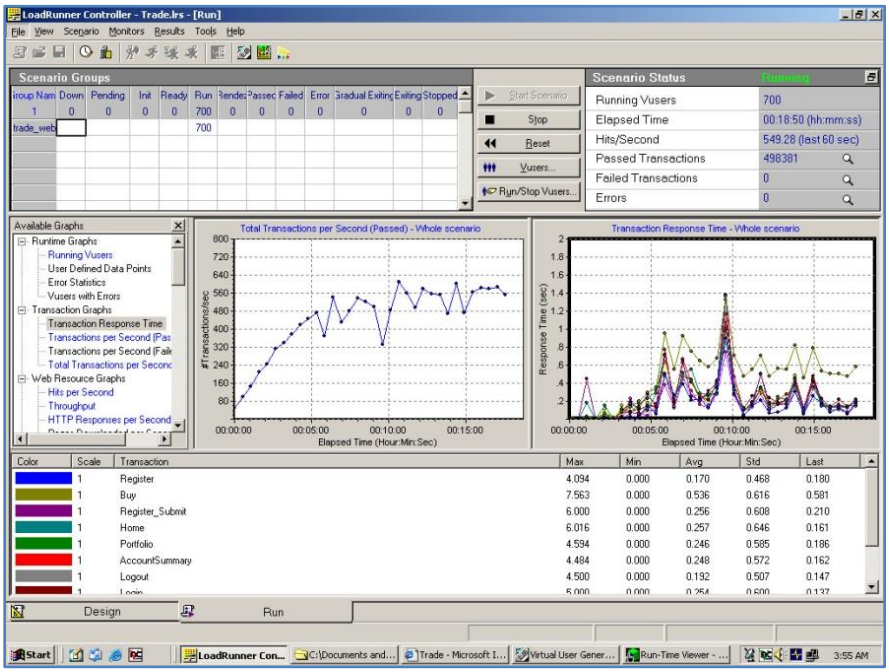

Figure 62: Mercury LoadRunner

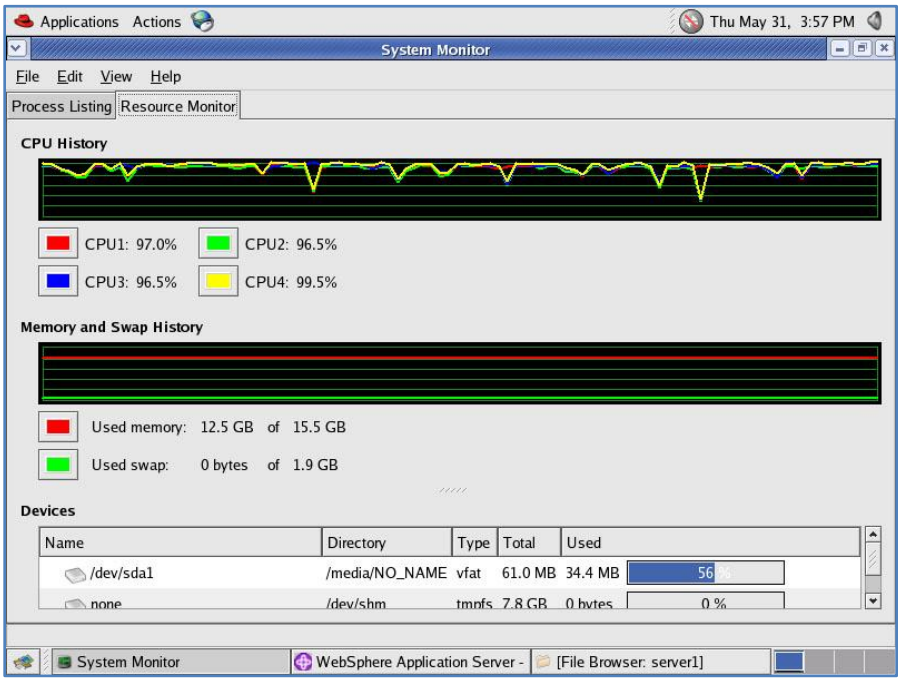

Figure 63: Application Server

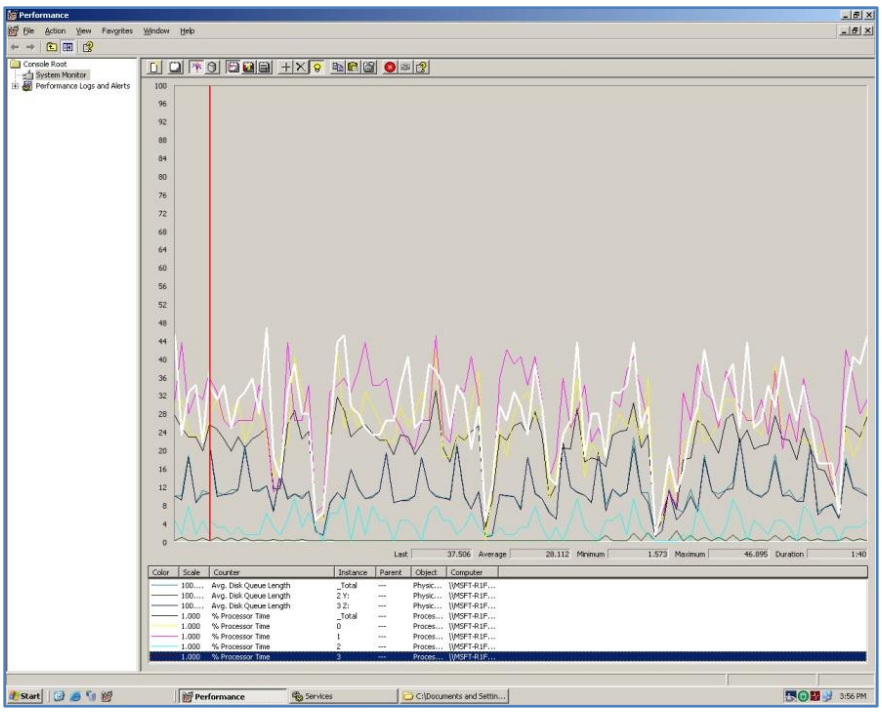

Figure 64: DB2

Messaging Benchmark Persistent Queue TwoPhase– Direct/JDBC Mode

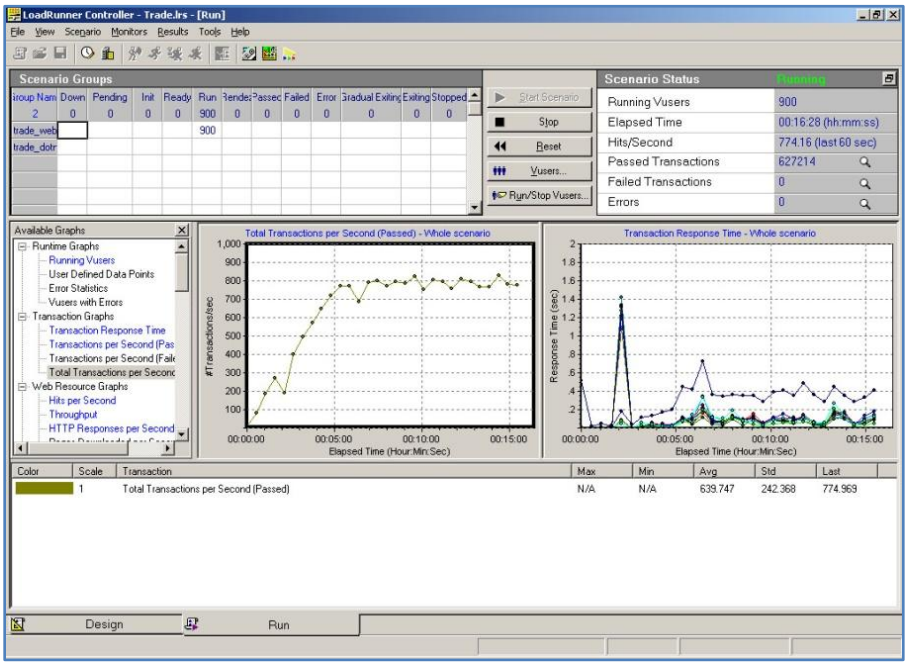

Figure 65: Mercury LoadRunner

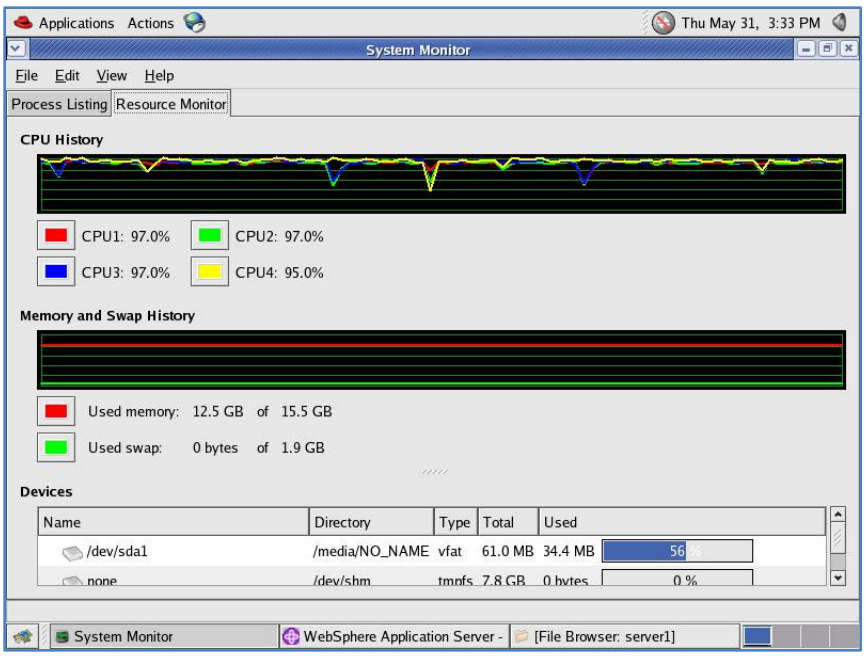

Figure 66: Application Server

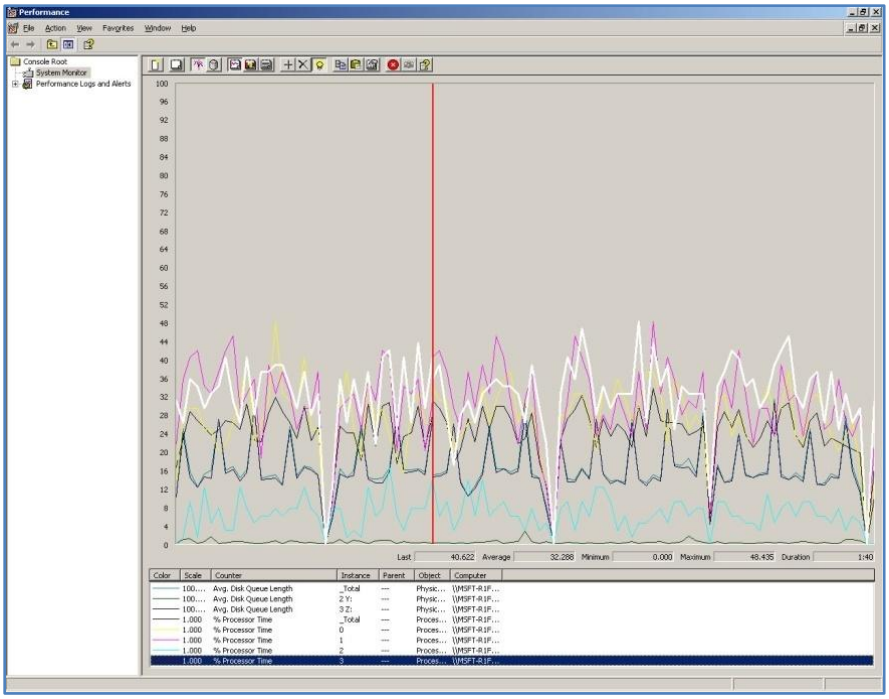

Figure 67: DB2

Messaging Benchmark NonPersistent Queue OnePhase – EJB Mode

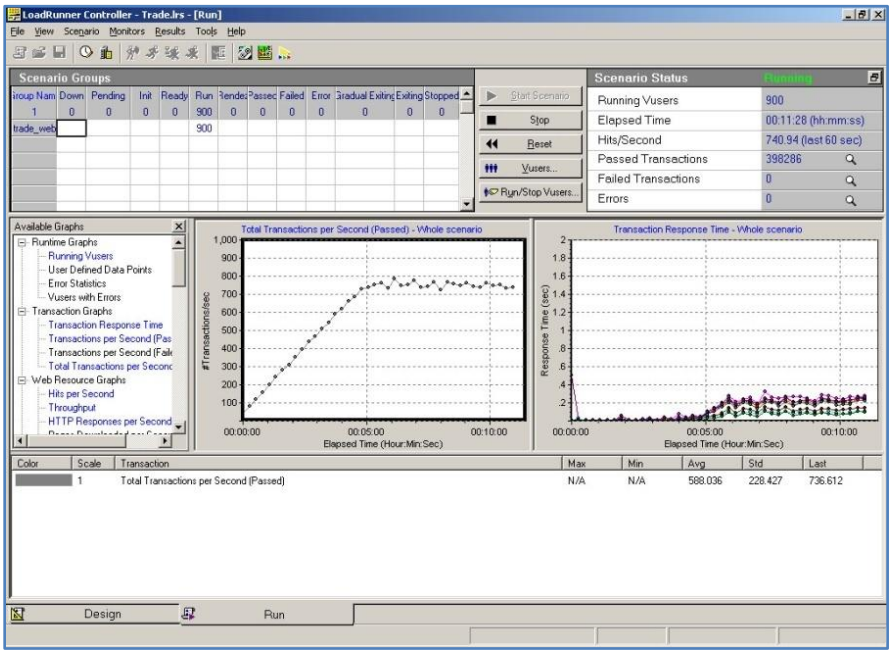

Figure 68: Mercury LoadRunner

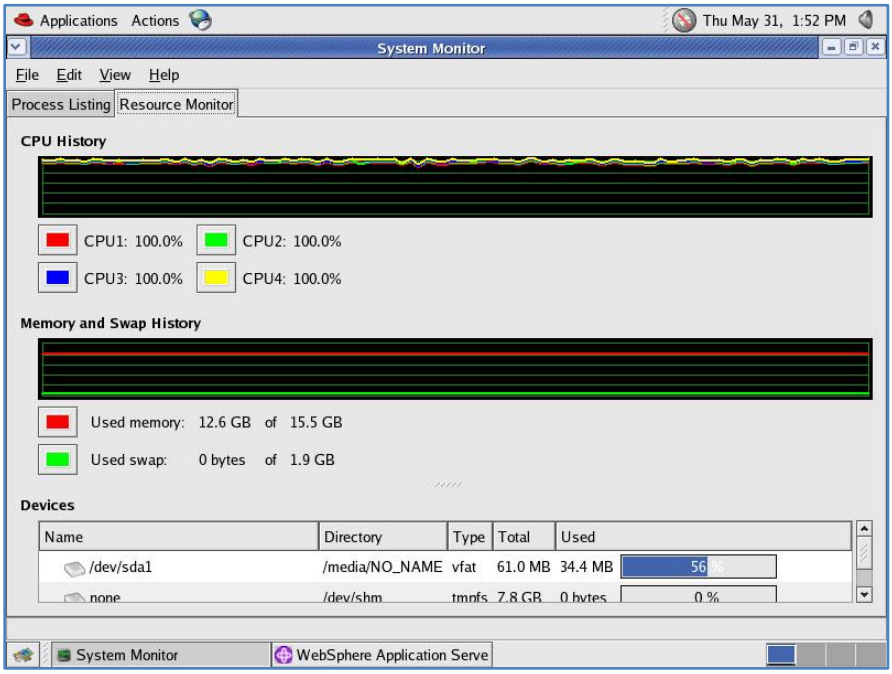

Figure 69: Application Server

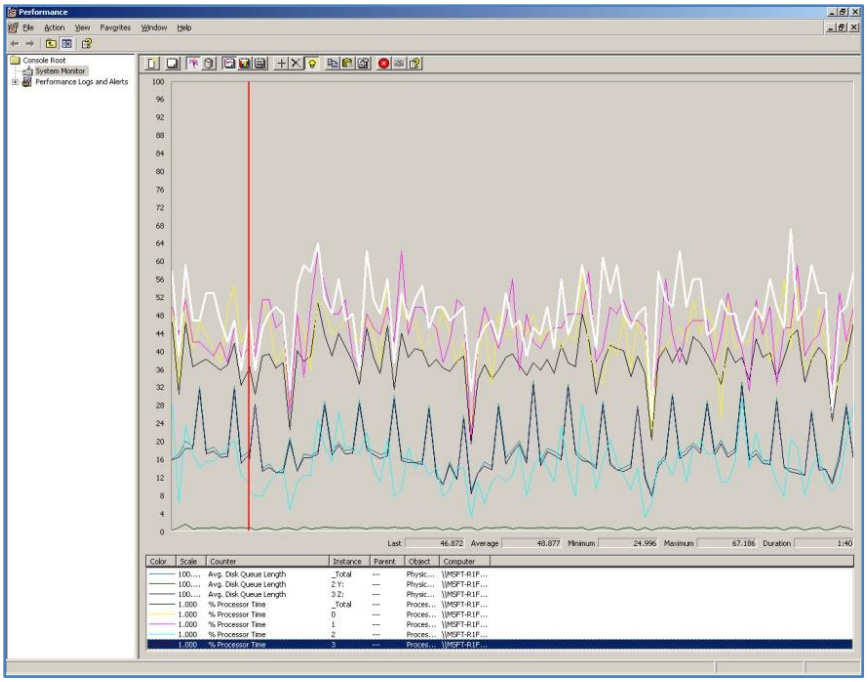

Figure 70: DB2

Messaging Benchmark NonPersistent Queue OnePhase Direct/JDBC Mode

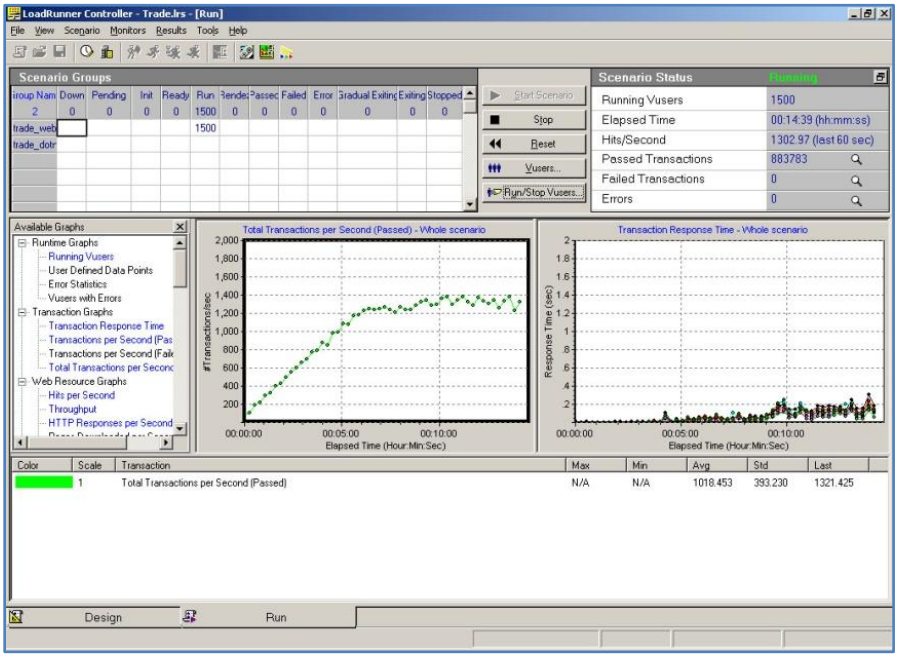

Figure 71: Mercury LoadRunner

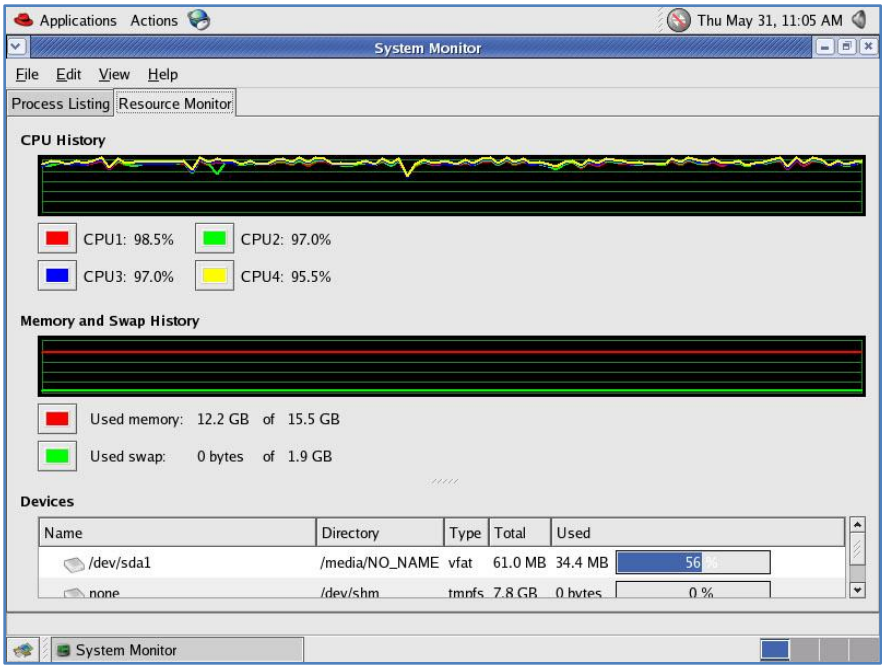

Figure 72: Application Server

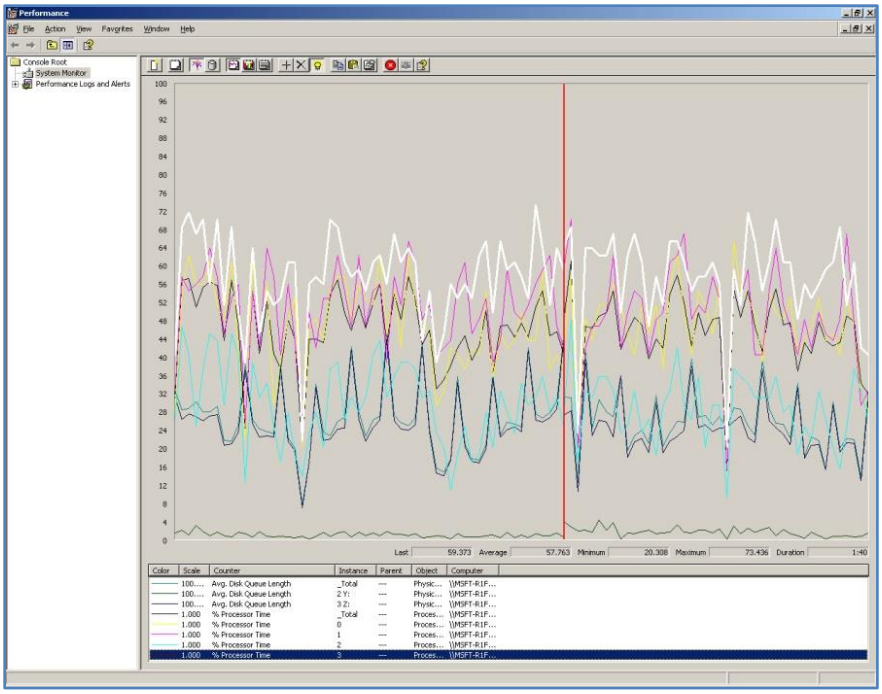

Figure 73: DB2

Monolithic Application: Synchronous Orders and Standard (non-remoted) Business Tier/Web Tier Access

EJB Mode

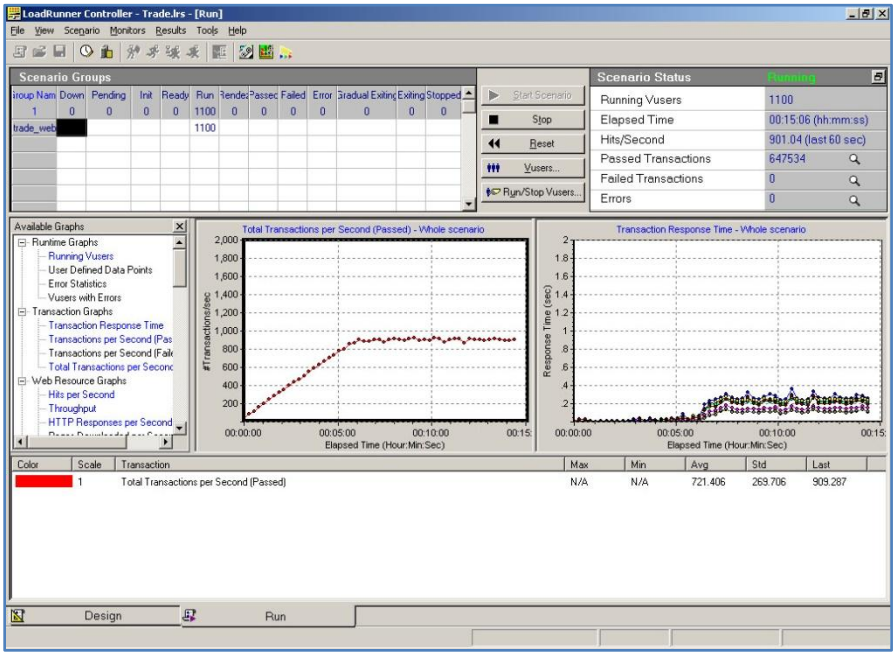

Figure 74: Mercury LoadRunner

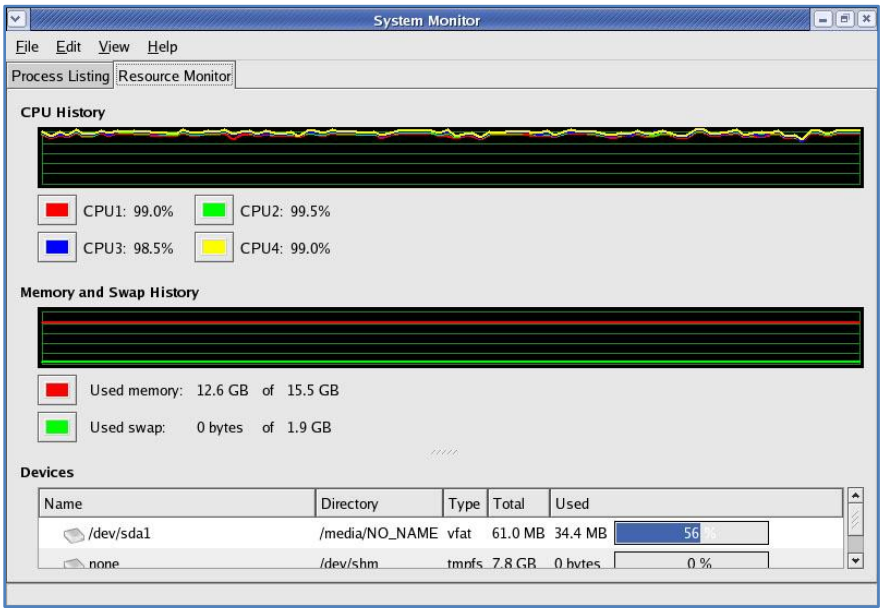

Figure 75: Application Server

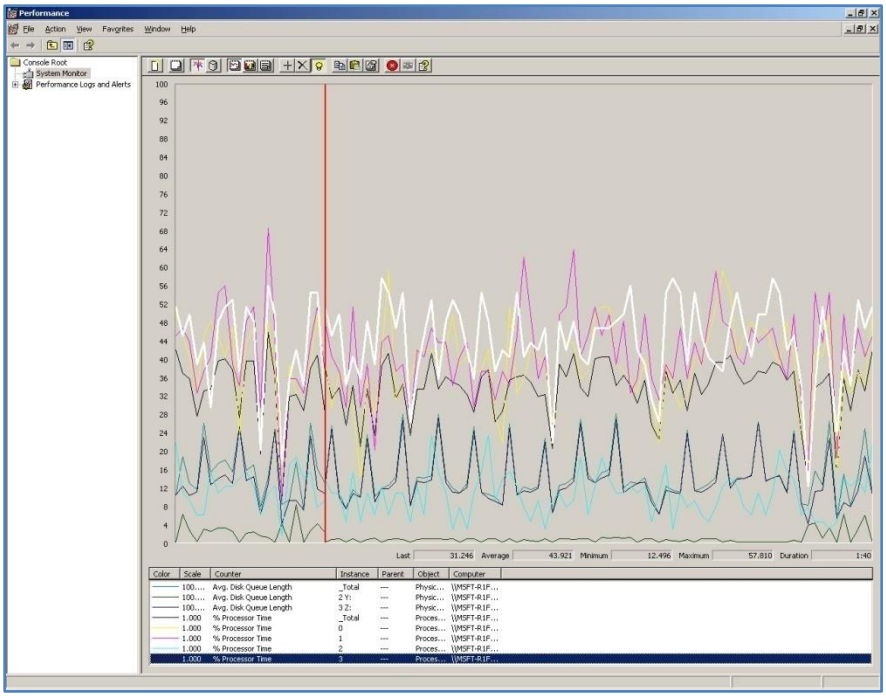

Figure 76: DB2

Direct Mode

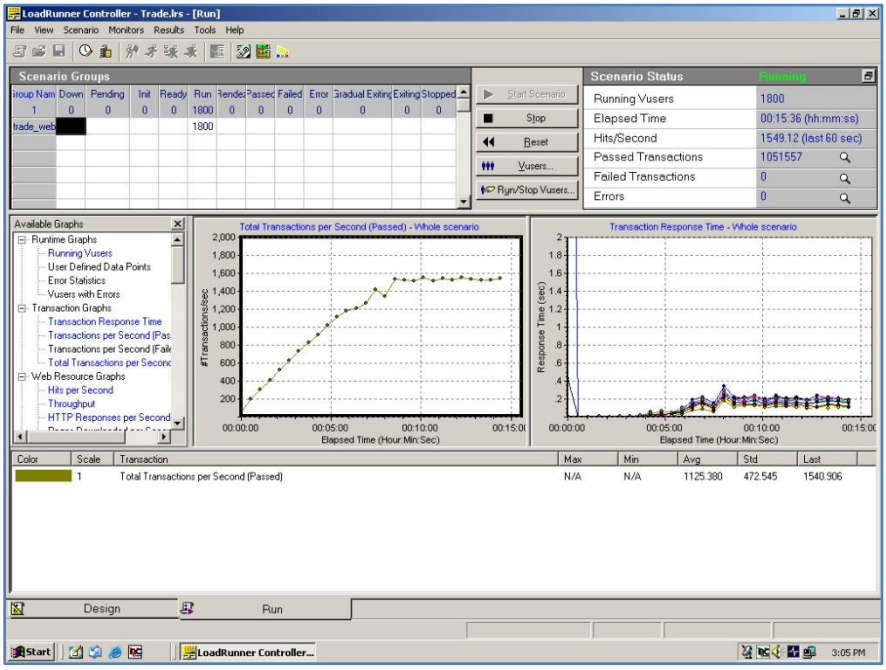

Figure 77: Mercury LoadRunner

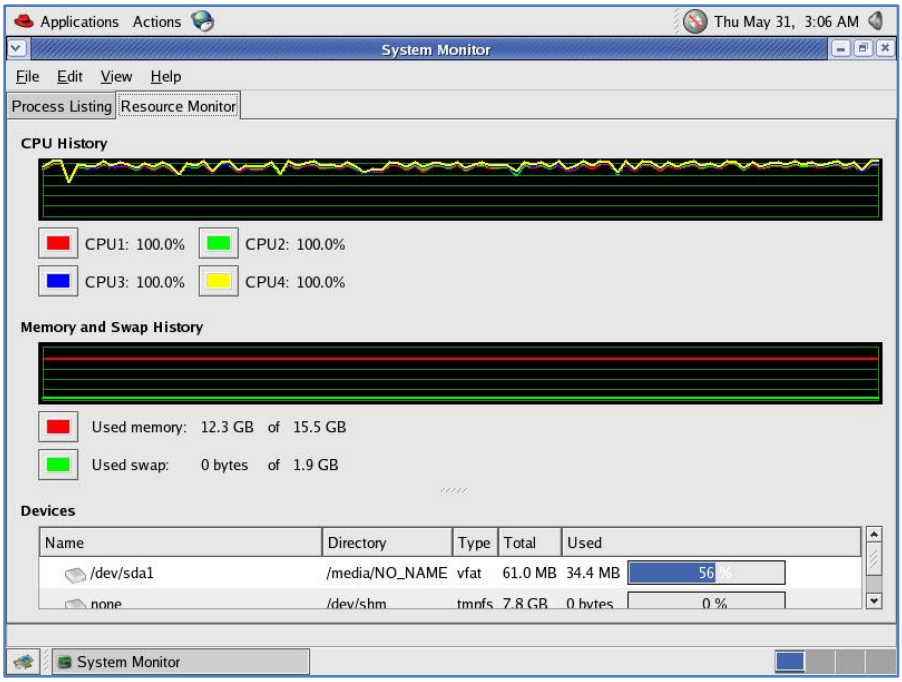

Figure 78: Application Server

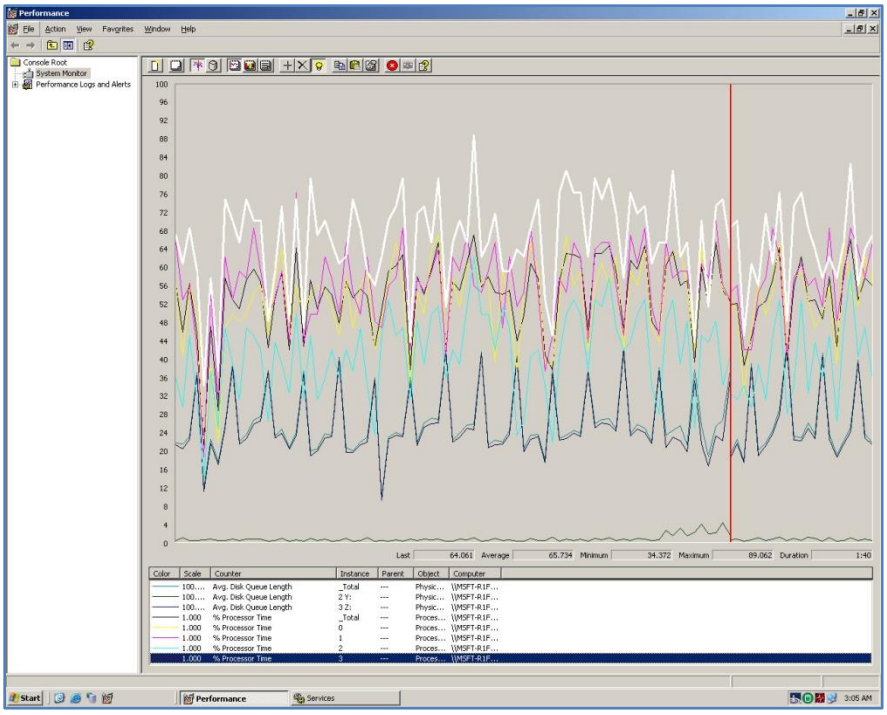

Figure 79: DB2

.NET 2.0/3.0

Web Services ASMX

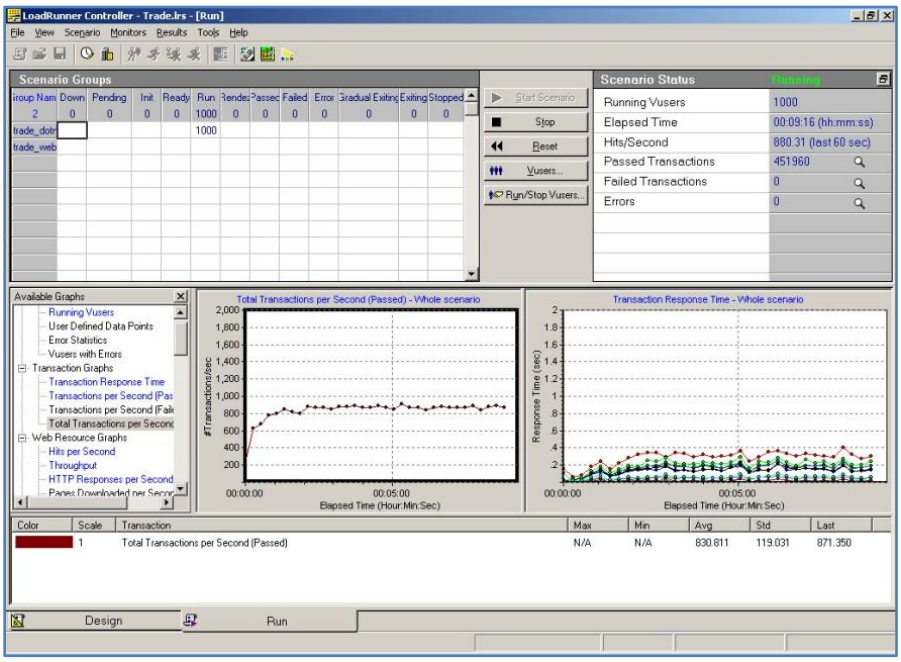

Figure 80: Mercury Load Runner

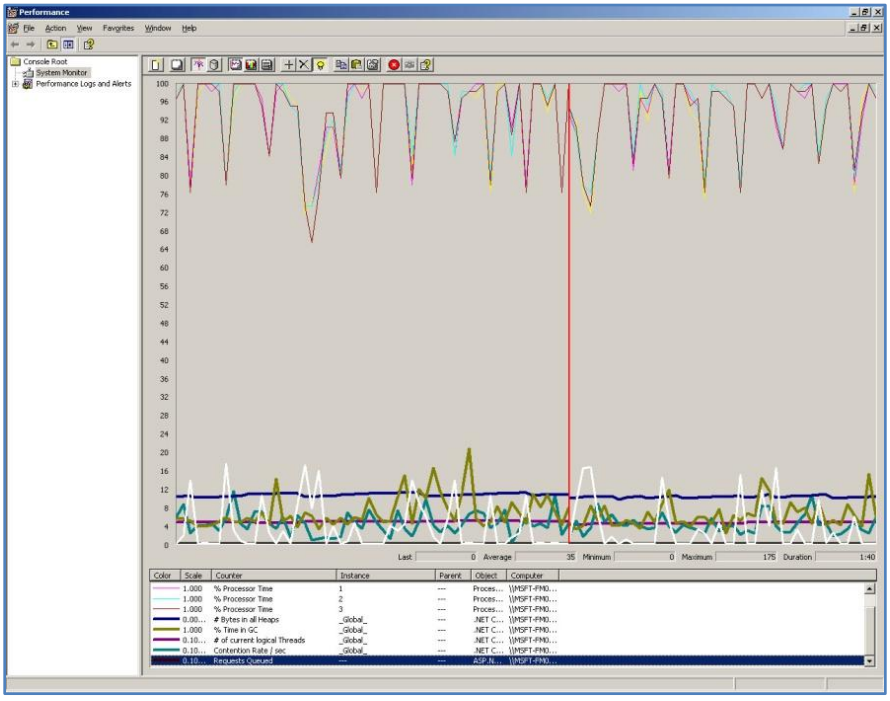

Figure 81: Application Server – Web Service Host

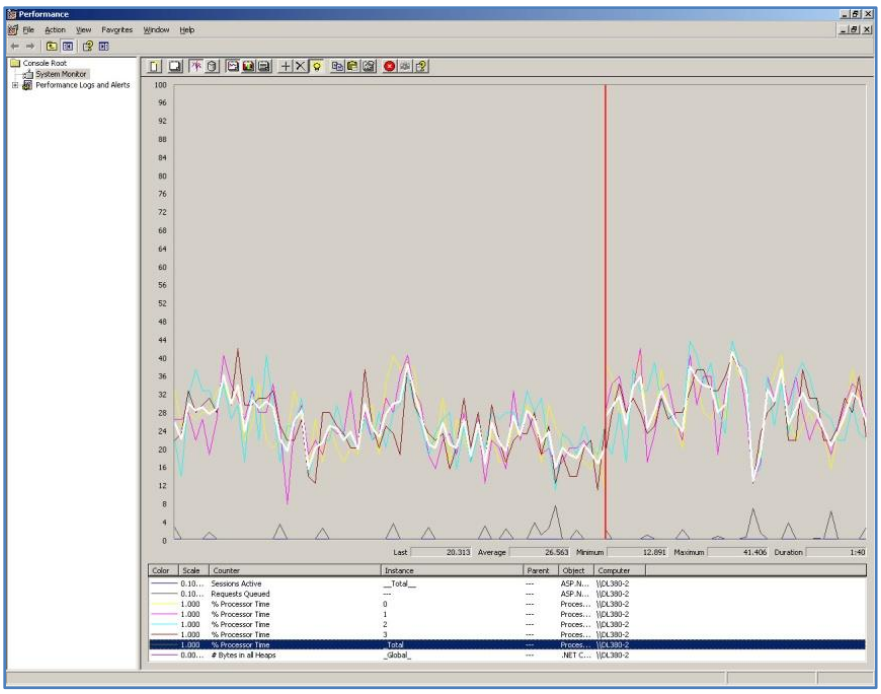

Figure 82: ASP.NET Server 1

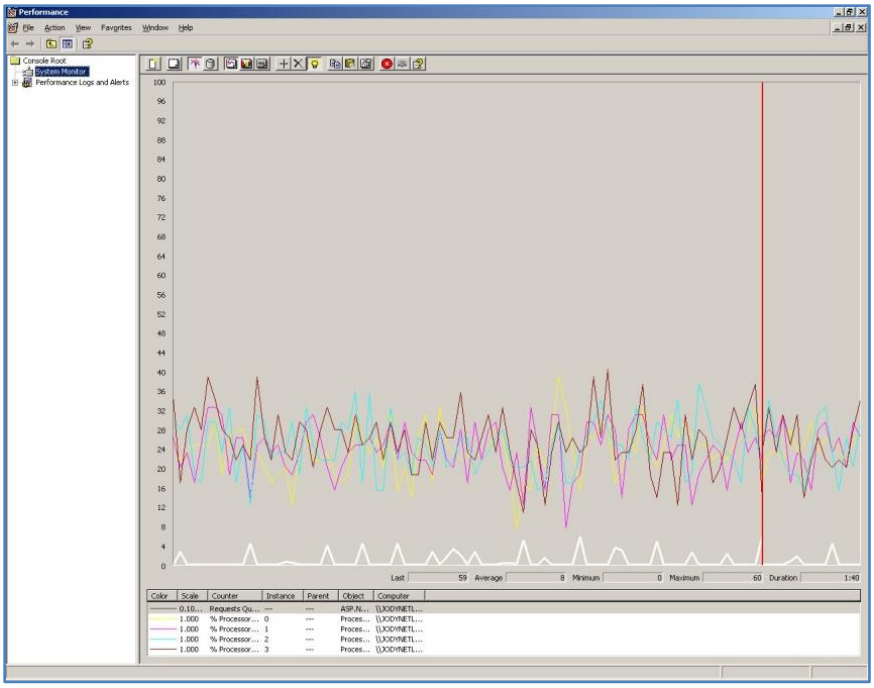

Figure 83: ASP.NET Server 2

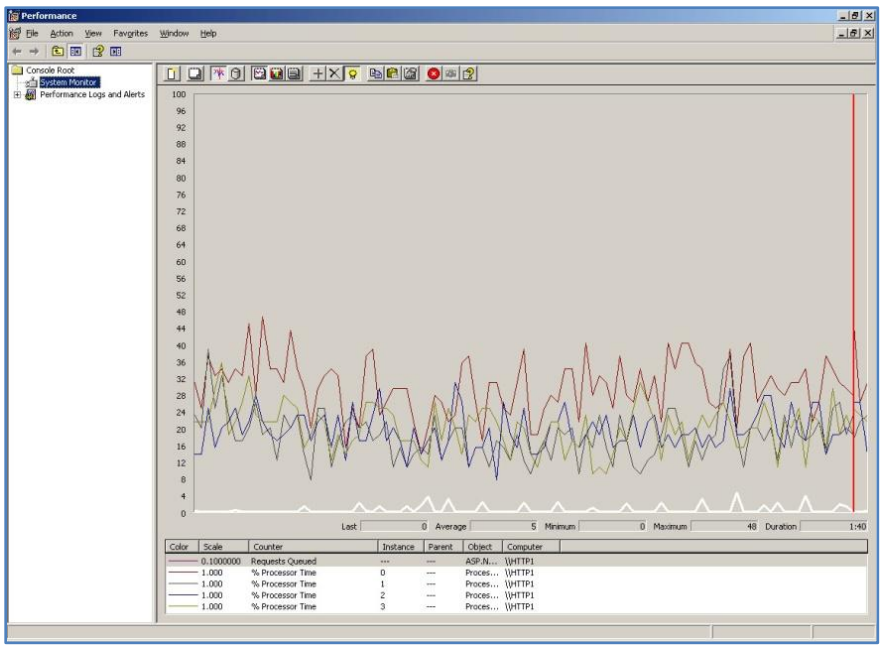

Figure 84: ASP.NET Server 3

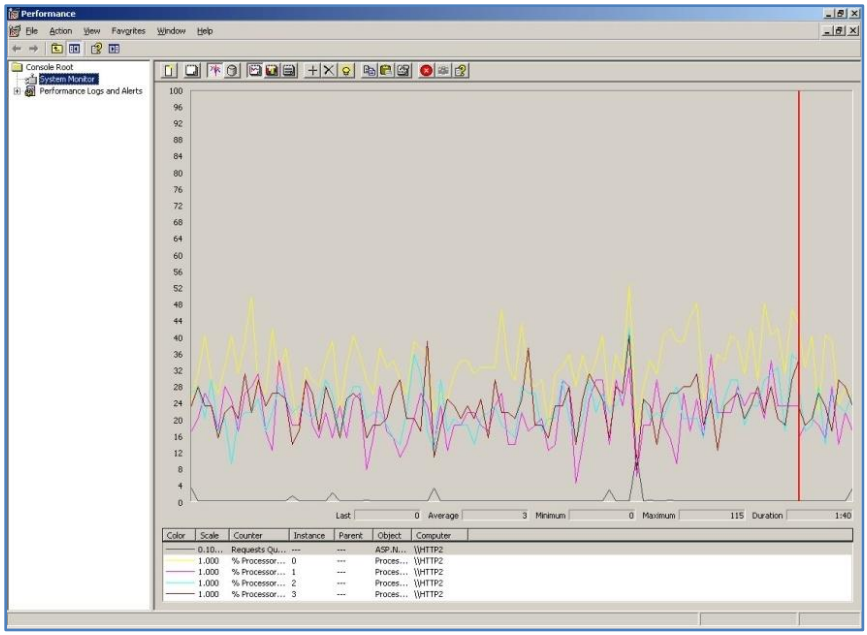

Figure 85: ASP.NET Server 4

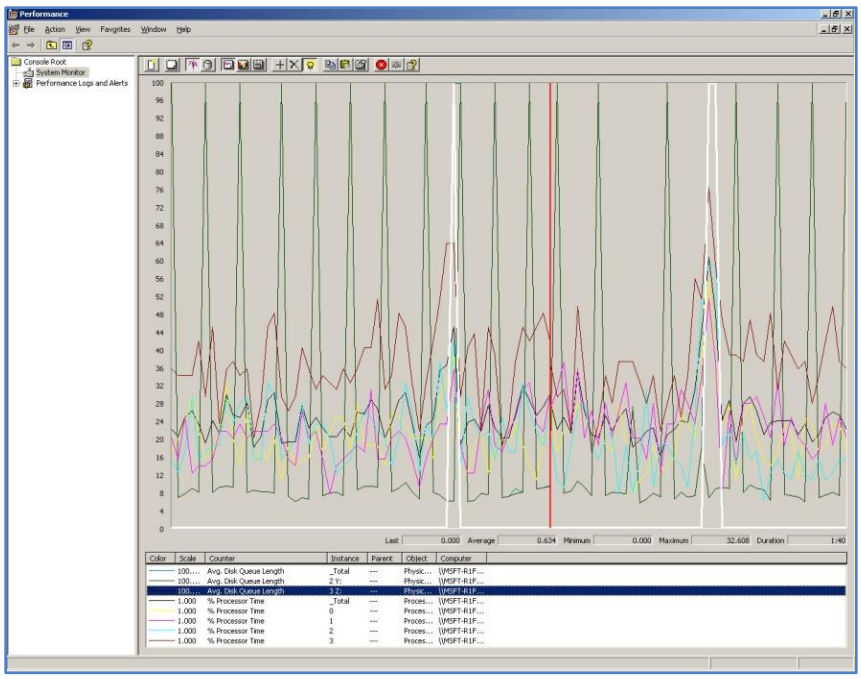

Figure 86: SQL Server 2005

WCF Web Service – IIS Hosted (basicHttpBinding)

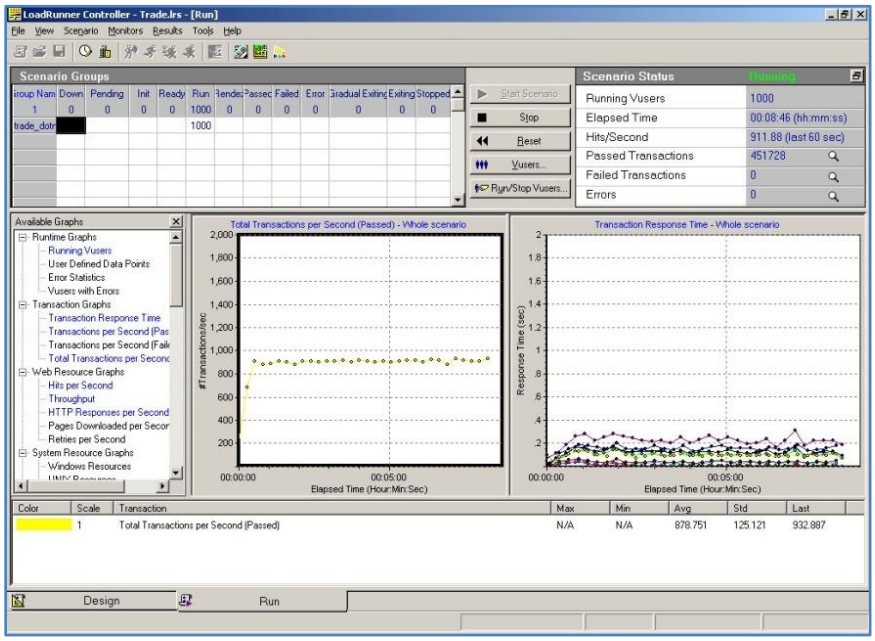

Figure 87: Mercury LoadRunner

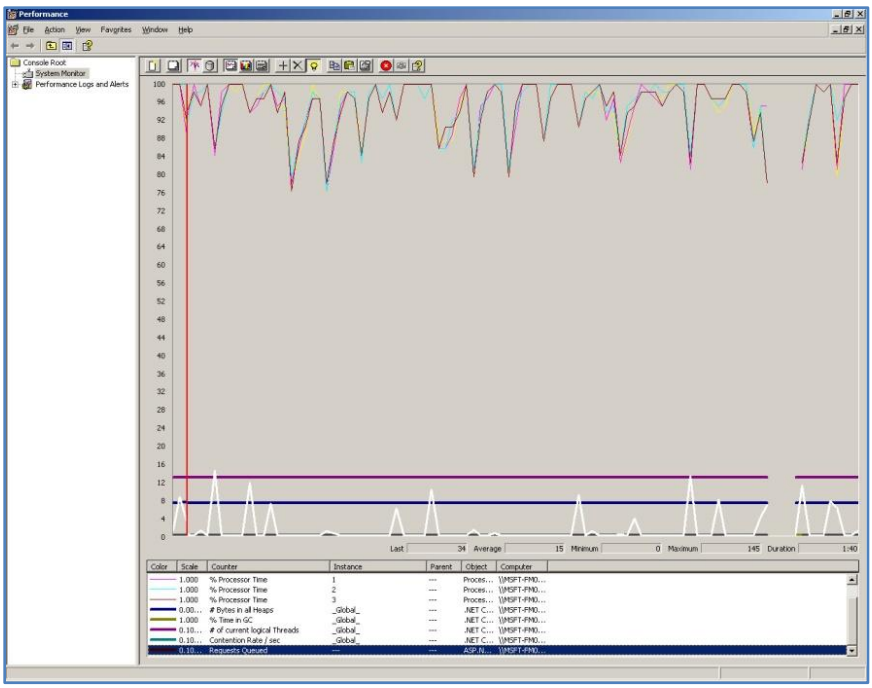

Figure 88: Application Server – Web Service Host

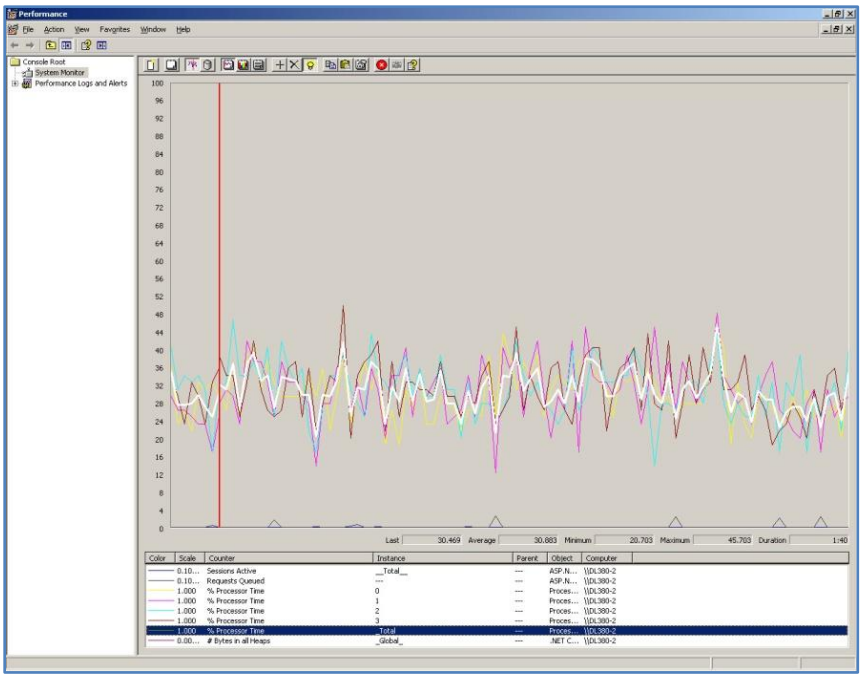

Figure 89: ASP.NET Server 1

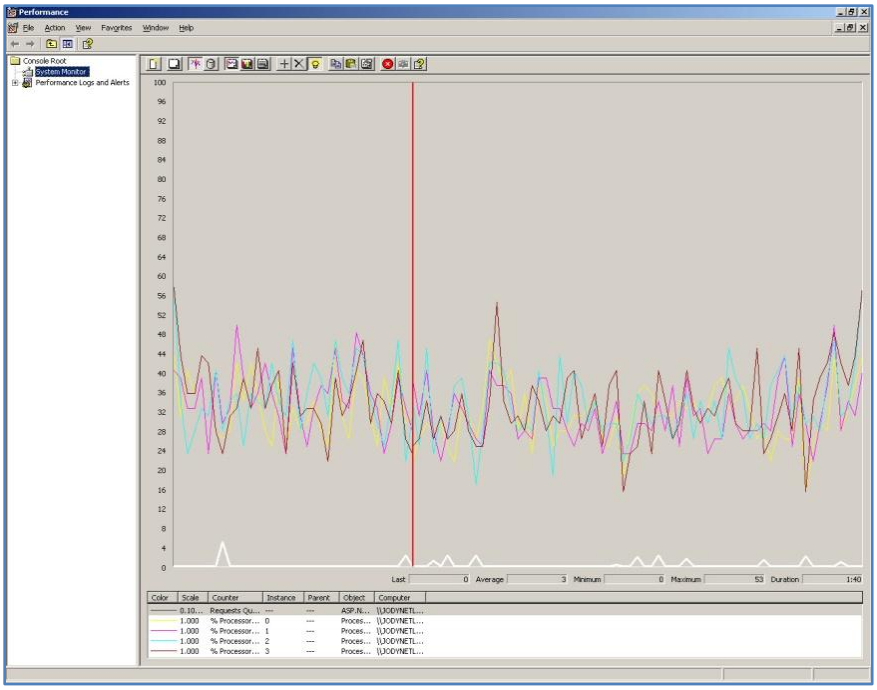

Figure 90: ASP.NET Server 2

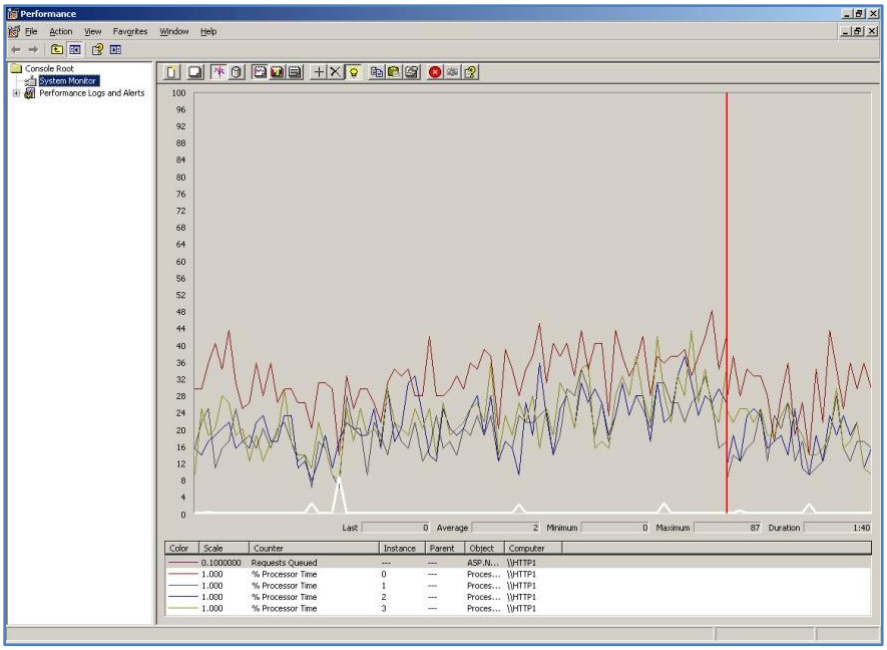

Figure 91: ASP.NET Server 3

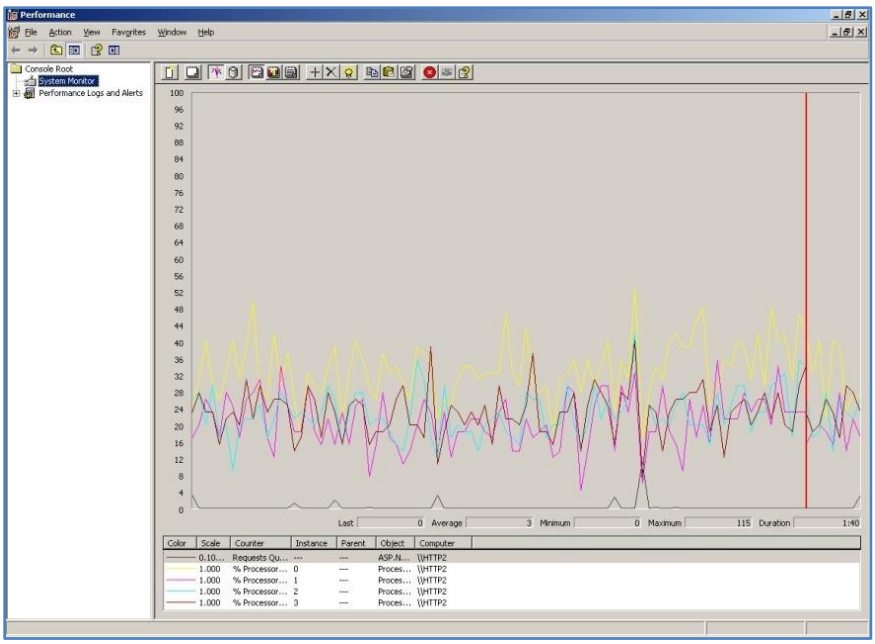

Figure 92: ASP.NET Server 4

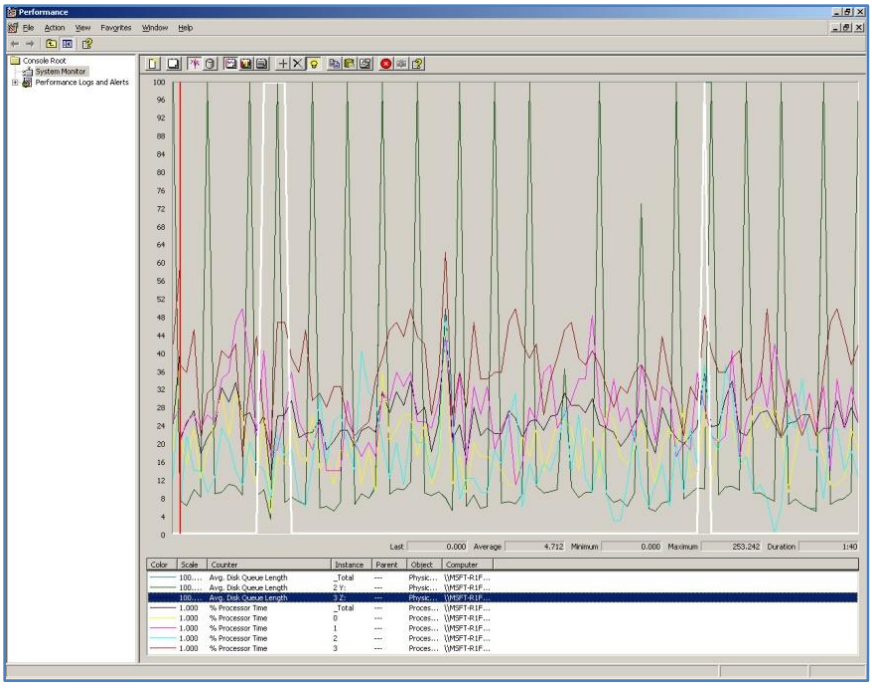

Figure 93: SQL Server 2005

WCF Web Service HTTP – Self Hosted (basicHttpBinding)

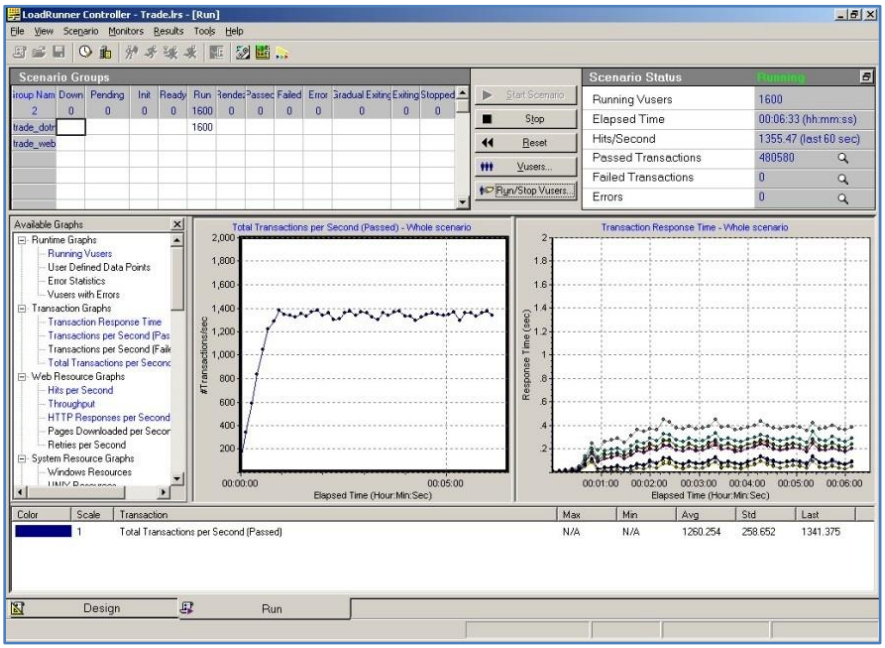

Figure 94: Mercury LoadRunner

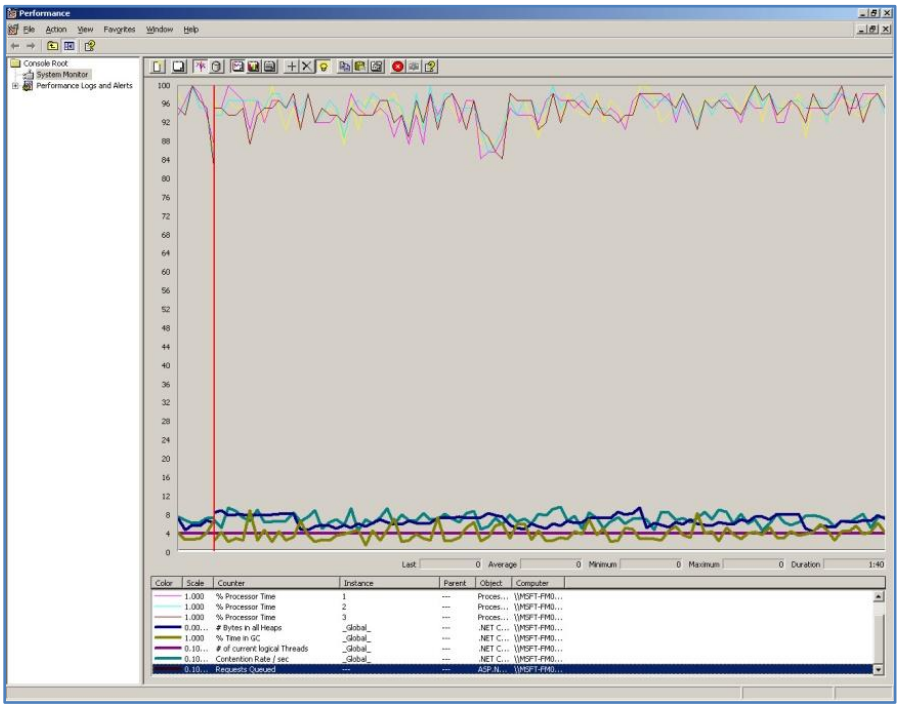

Figure 95: Application Server- Web Service Host

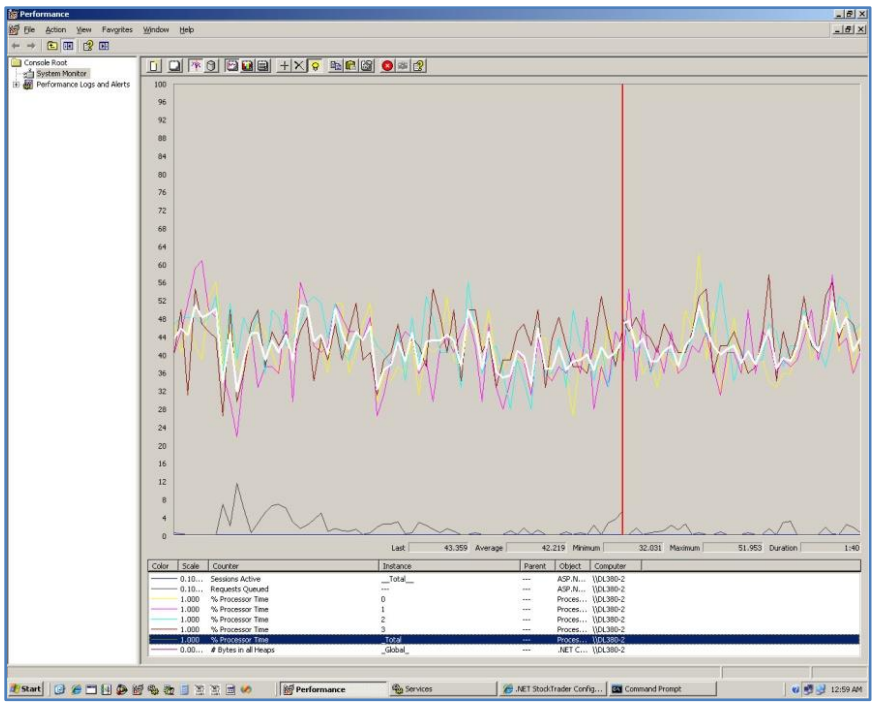

Figure 96: ASP.NET Server 1

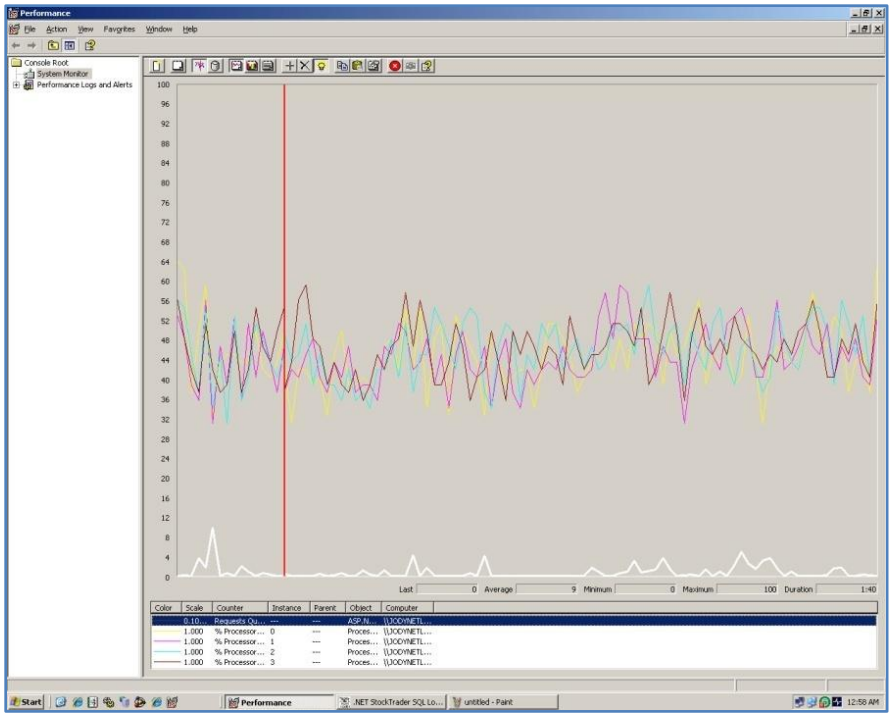

Figure 97: ASP.NET Server 2

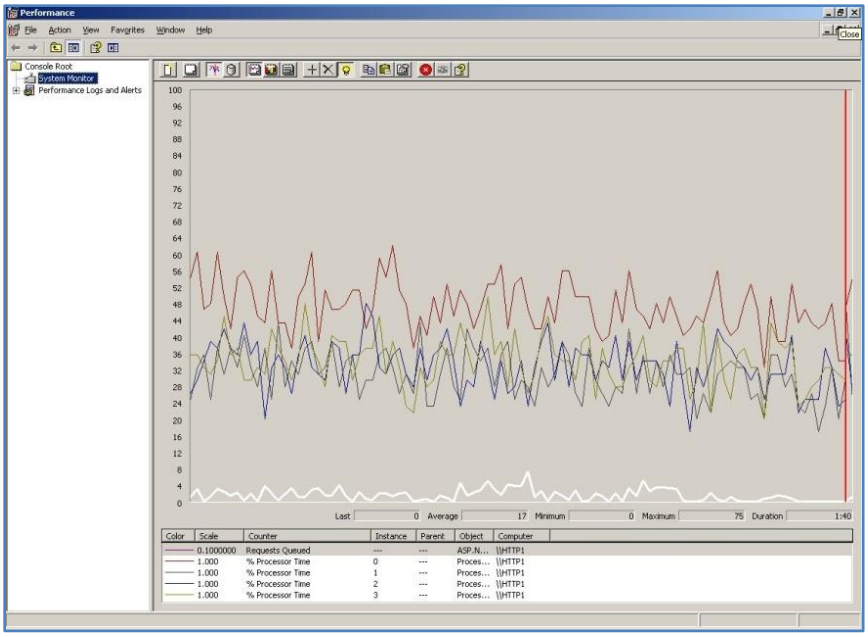

Figure 98: ASP.NET Server 3

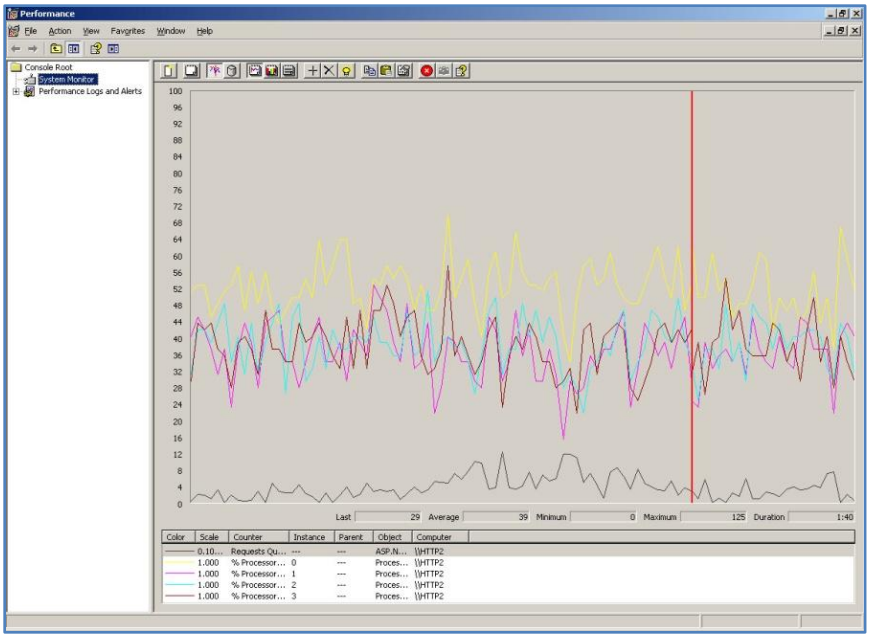

Figure 99: ASP.NET Server 4

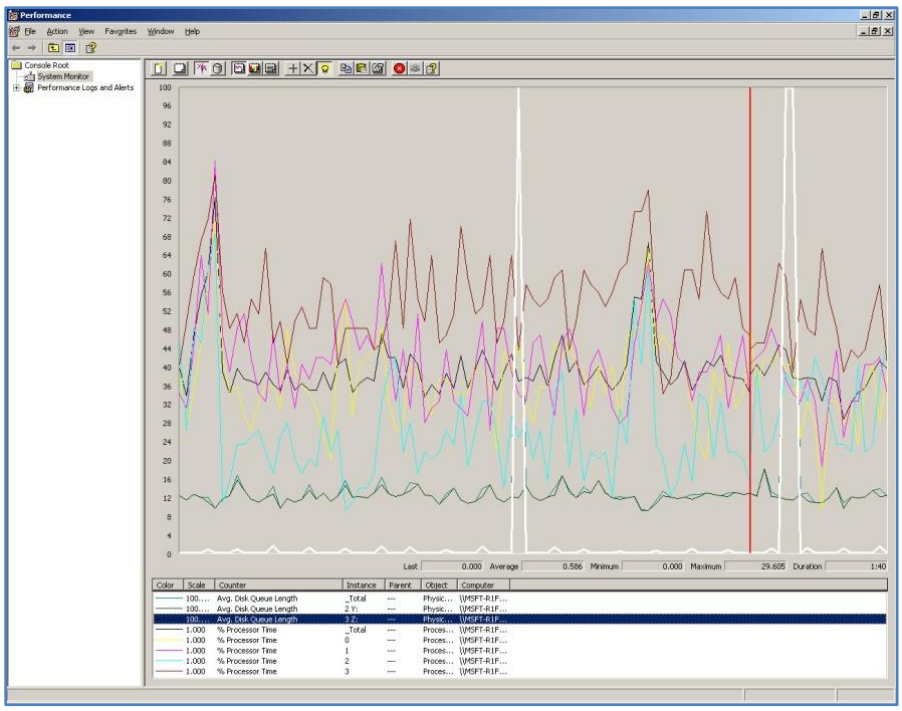

Figure 100: SQL Server 2005

WCF Web Service TCP/Binary – Self Hosted (netTcp Binding)

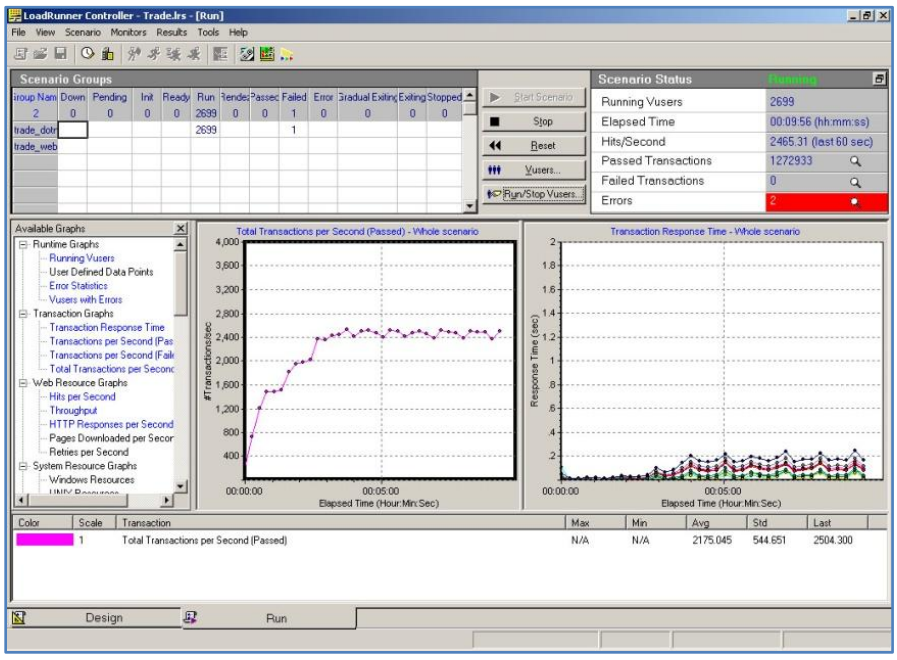

Figure 101: Mercury LoadRunner

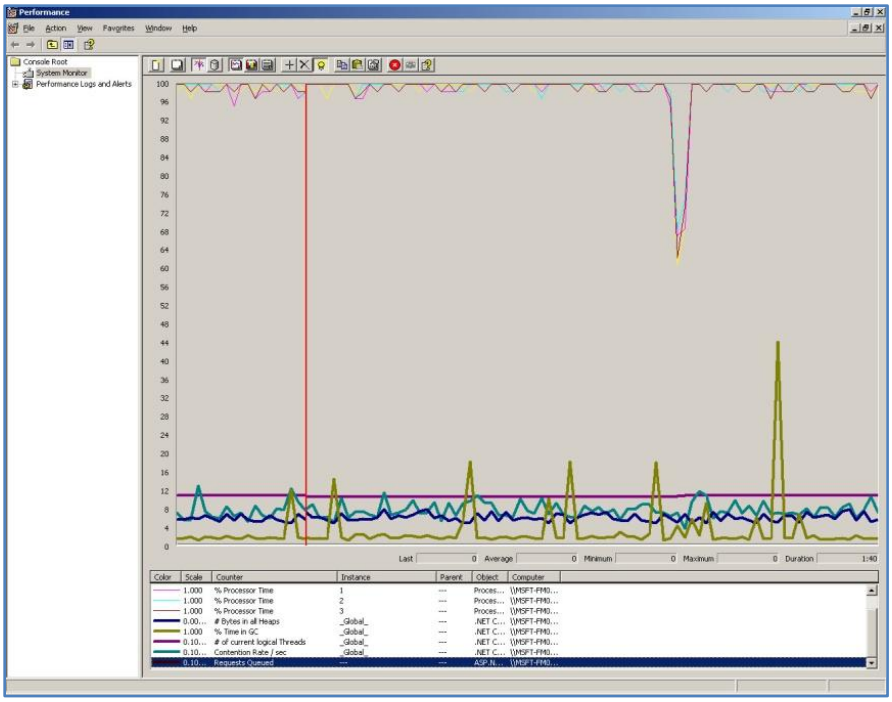

Figure 102: Application Server – Web Service Host

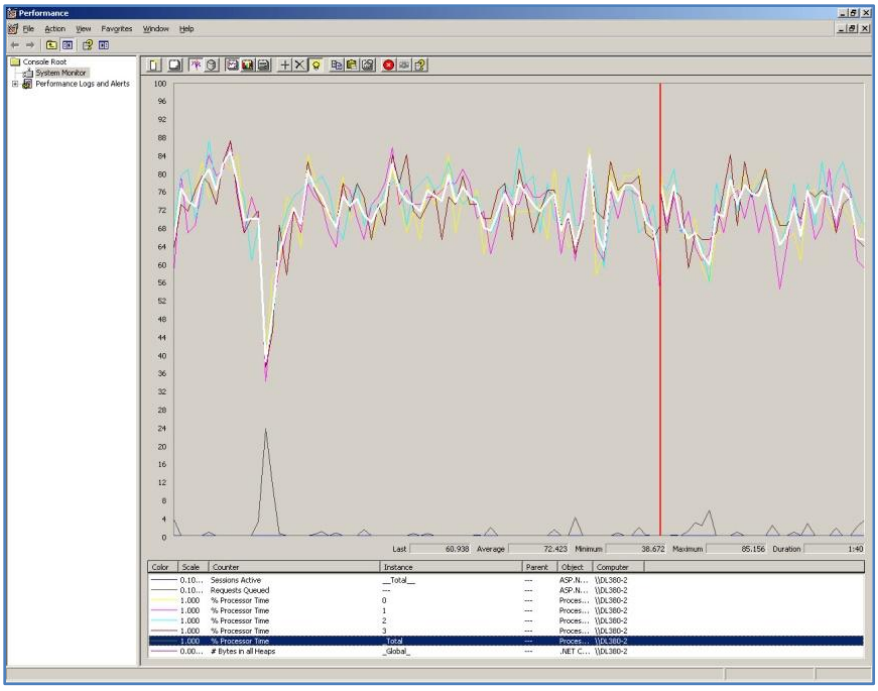

Figure 103: ASP.NET Server 1

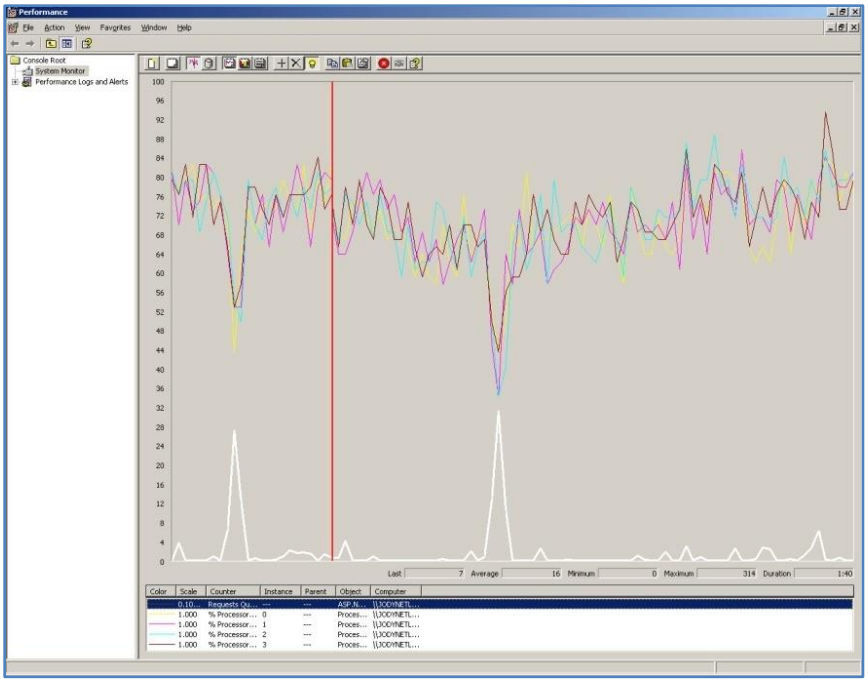

Figure 104: ASP.NET Server 2

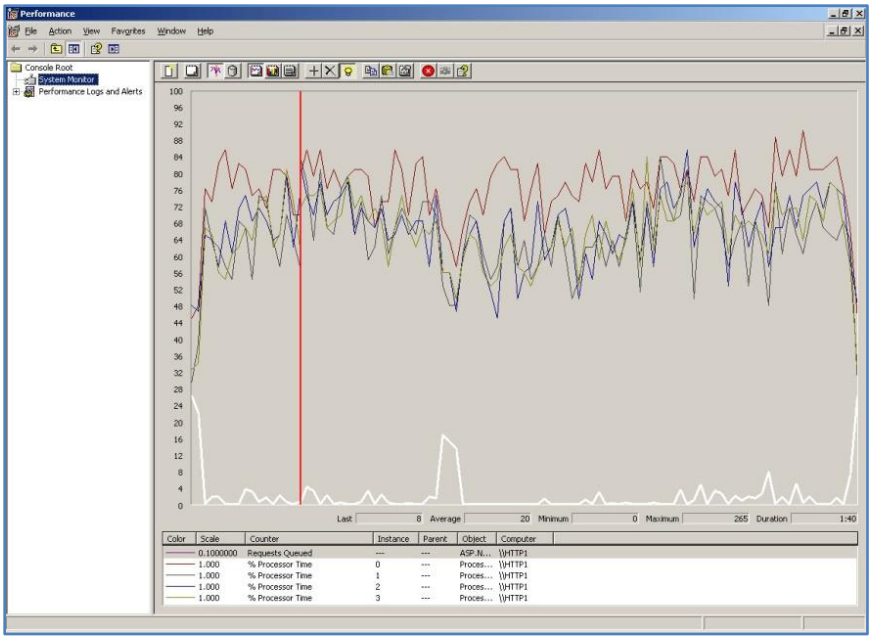

Figure 105: ASP.NET Server 3

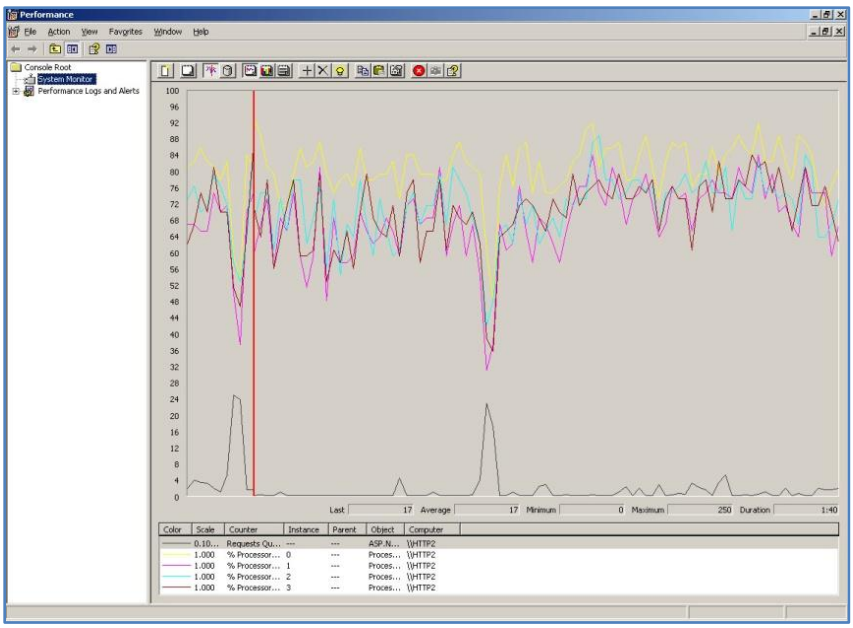

Figure 106: ASP.NET Server 4

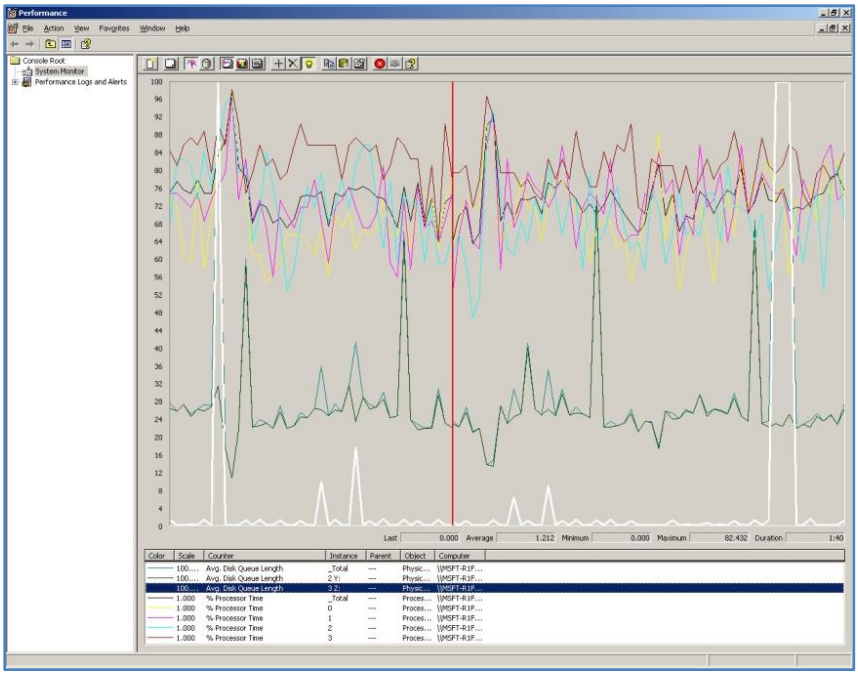

Figure 107: SQL Server 2005

Messaging Benchmark Persistent Queue- TwoPhase (WCF over transacted/durable MSMQ)

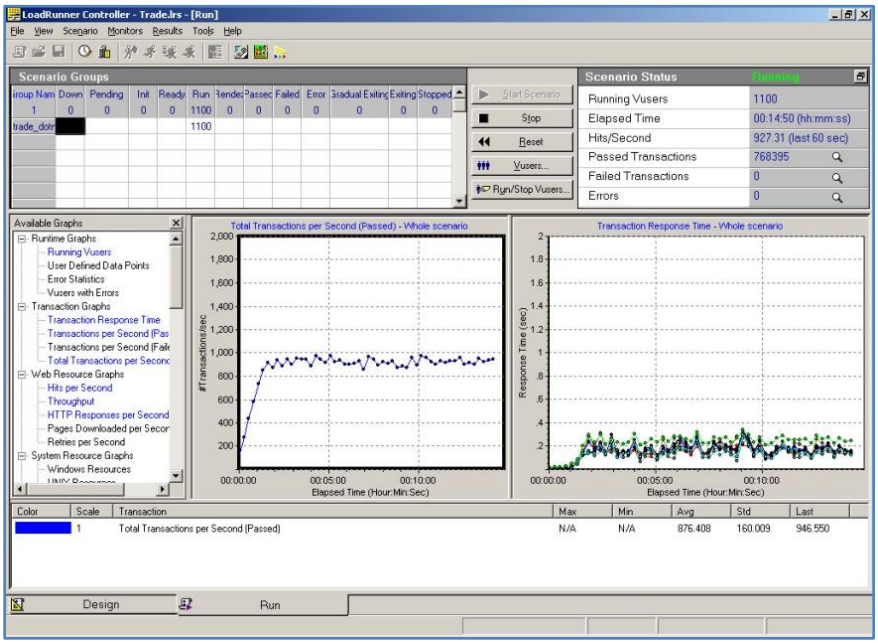

Figure 108: Mercury LoadRunner

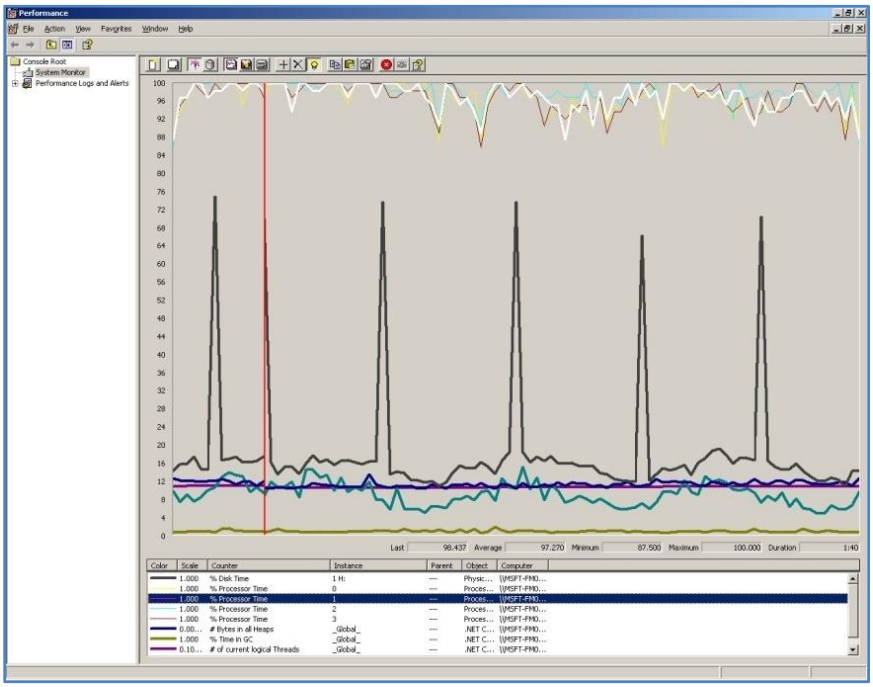

Figure 109: Application Server

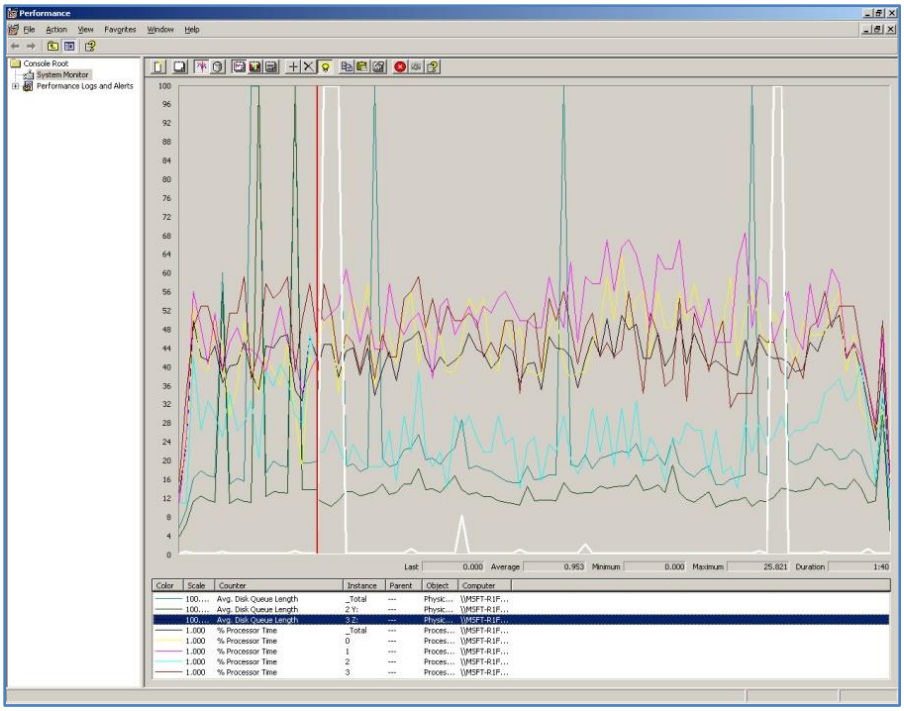

Figure 110: SQL Server 2005

Messaging Benchmark Non-Persistent Queue- OnePhase (WCF over non-transacted/in-memory MSMQ)

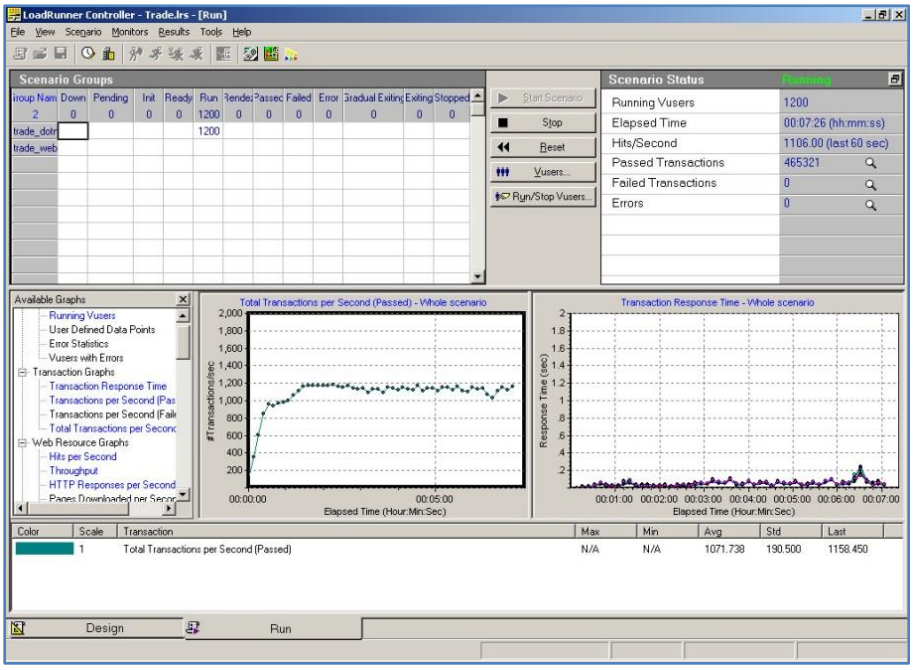

Figure 111: Mercury LoadRunner

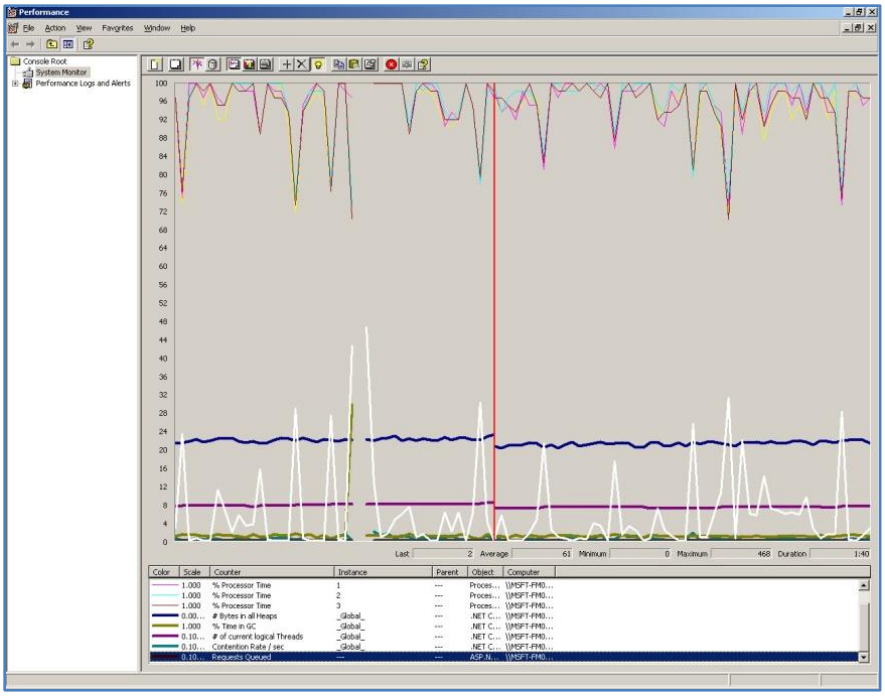

Figure 112: Application Server

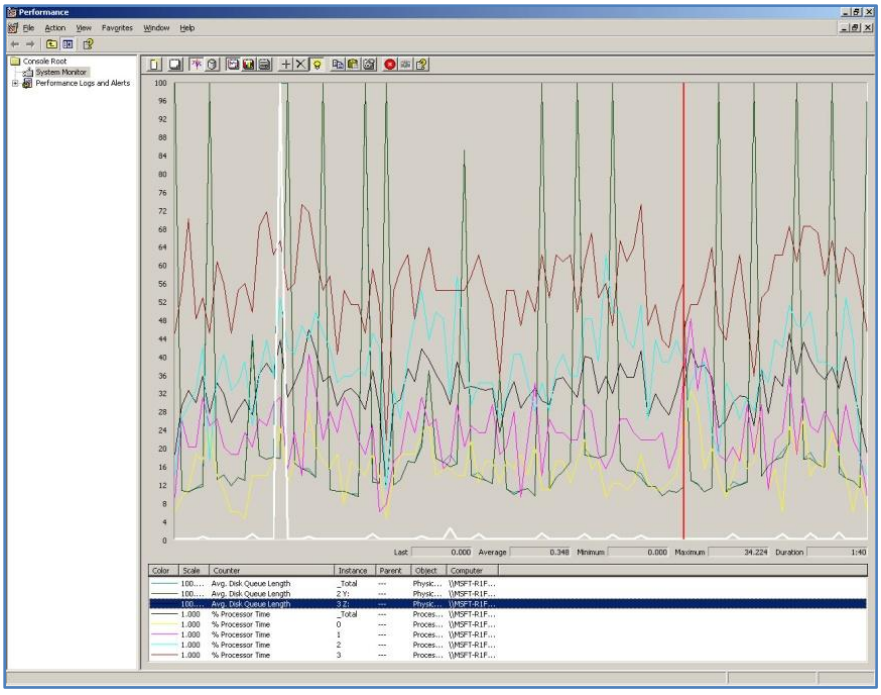

Figure 113: SQL Server 2005

Monolithic Application: InProcess/Synchronous Orders

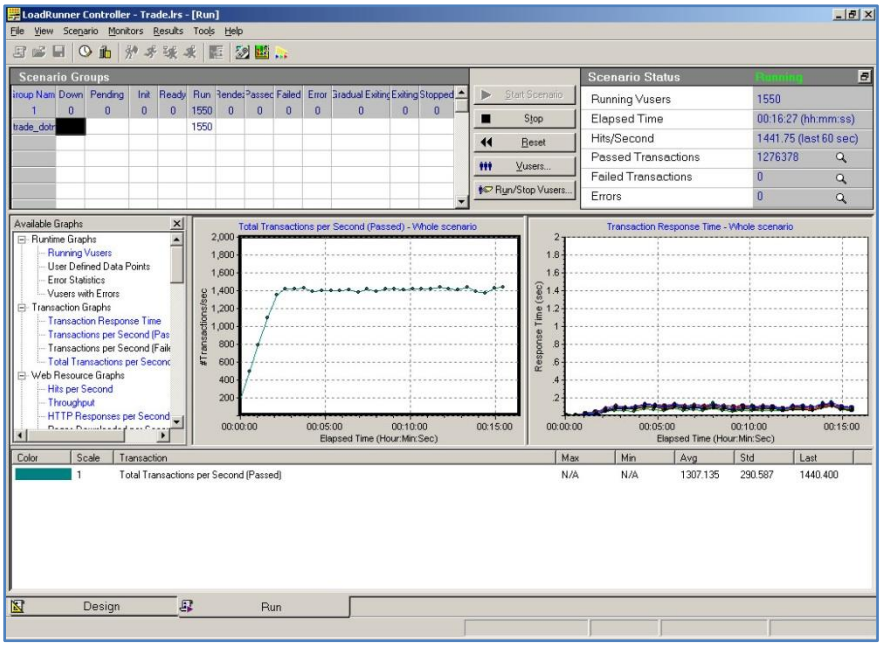

Figure 114: Mercury LoadRunner

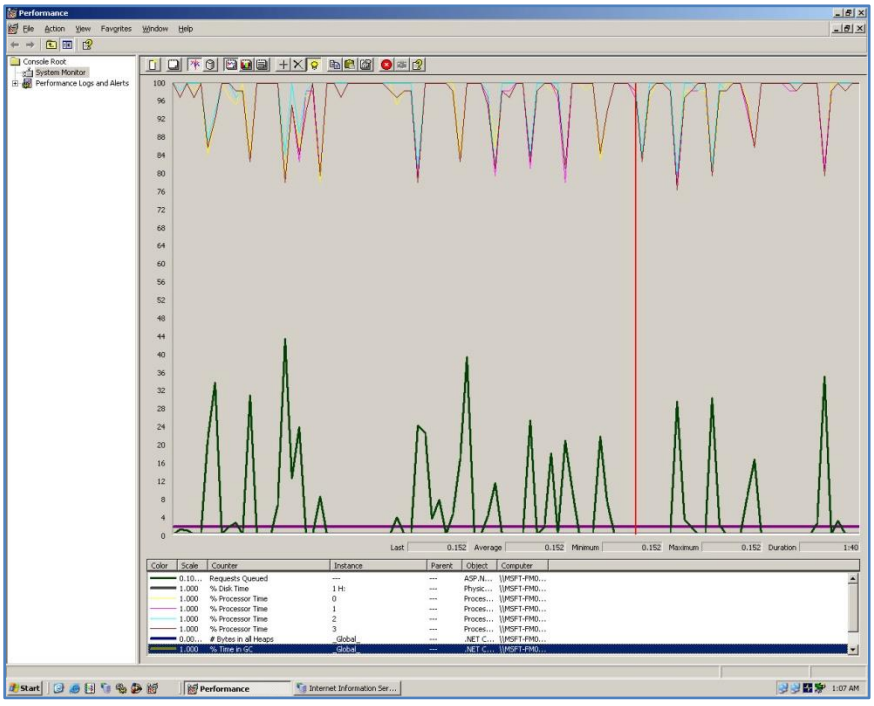

Figure 115: Application Server

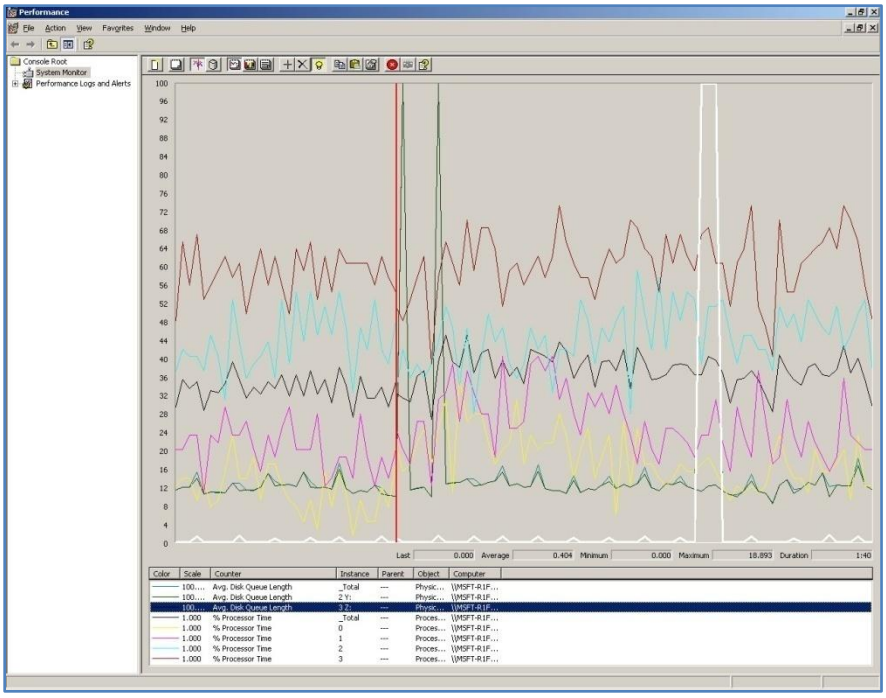

Figure 116: SQL Server 2005

Appendix C: Tuning Parameters

Linux OS Tuning

net.ipv4.tcp_max_syn_backlog=1024

kernel.msgmni=1024

kernel.sem=1000 32000 32 512

fs.file-max=65535

kernel.shmmax =4294967295

net.core.netdev_max_backlog = 20000

net.core.somaxconn = 20000

net.ipv4.tcp_fin_timeout = 30

net.ipv4.tcp_syn_retries = 20

net.ipv4.tcp_synack_retries = 20

net.ipv4.tcp_sack = 0

net.ipv4.tcp_timestamps = 0

net.ipv4.conf.all.arp_ignore = 3

net.ipv4.conf.all.arp_announce = 2

Open File Handle limit (soft) increased to 20000

Windows OS Tuning

No tuning was required on the core Windows Server OS for either application server platform.

WebSphere Tuning

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

Trade 6.1 Configured not to write System.Out messages

EJB Cache Size = 20000

HTTP Channel maximum persistent requests = -1

Minimum Web Container threads = 100

Maximum Web Container threads = 100

Minimum ORB threads = 80

Maximum ORB threads = 80

Minimum Default threads = 20

Maximum Default threads = 20

Minimum Message Listener Service Threads = 80

Maximum Message Listener Service Threads = 80

Minimum SIBInBound Thread = 80

Maximum SIBInbound Thread = 80

Minimum SIBFAPThread = 60

Maximum SIBFAPThread = 60

Custom JavaEnvironment Variable: com.ibm.websphere.ejbcontainer.poolsize value = "*=75,750"

SIB Bus Security = Disabled

Discard Messages = on

Hi Message Threshold = 50000

Quality of Service/Persistent = Assured Reliable

Quality of Service/Non Persistent = Express/Non Persistent

ReadAhead for Queue enabled

MaxConcurrency/Max Endpoints for Queue = 20

MaxBatchSize for JMS/Messaging = 5

Minimum JDBC Connections in Pool = 90

Maximum JDBC Connections in Pool = 90

Minimum Queue Connection Factory Connections in Pool = 90

Maximum Queue Connection Factory Connections in Pool = 90

EJB Pass By Reference On; configured to use ORB Thread pool as recommended by IBM

Java Heap Size: Windows = 1540 MB (maximum for 32-bit on Windows)

Java Heap Size: Linux = 2000 MB (maximum for 32-bit on Linux)

All runs: Trade 6.1 configured with "Enable Long Run Support" off to ensure it properly displays orders on the Account Page. With our large database load, we never noticed any perf degradation over a 30 minute measurement interval.

IBM HTTP Server Windows Tuning

Access Log Off

Max KeepAlive Requests 3000

2048 Max threads

2048 Threads/child

IBM HTTP Server Linux Tuning

Access Log Off

Max KeepAlive Requests 3000

ThreadLimit 50

ServerLimit 64

StartServers 50

MaxClients 3200

MinSpareThreads 100

MaxSpareThreads 100

Threads/Child 50

MaxRequests/Child 0

.NET 2.0/3.0 Tuning

.NET Worker Process

Rapid Fail Protection off

Pinging off

Recycle Worker Process off

ASP.NET

Authentication set to "None" to match anonymous access of IBM WebSphere Trade 6.1

Forms Authentication Timeout=5 minutes

IIS 6.0 Virtual Directory

Authentication Basic Only

Access Logging Off

Windows Communication Foundation/ASMX Web Services

ServicePointManager.DefaultConnectionLimit = 64

(note, this is a key setting for Web Service clients running under load. Without this setting, Web Service clients (our four ASP.NET App Servers) will be throttled to 2 network connections per outbound IP Address. This is set programmatically, although it

WCF basicHttp, nNetTcp and Msmq bindings: Security = "None" (no transport security for Web services or the Service Integration Bus is configured for Trade 6.1 as well, see tuning for WebSphere)

Service Behavior for Business Services and Order Processor Service:

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

Order Processor Service: maxBatchSize = 5 (set programmatically in the Host)

.NET StockTrader

Max DB Connections = 90

Min DB Connections = 90

MSMQ

Connection Caching turned on

MSTDC

Transaction Timeout = 15 seconds

Network DTC Access Turned on (inbound and outbound allowed)

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 = 150

SQL/Server

Logging files expanded to 15 GB

Logging Set to One Drive Array (array a)

Database file on Second Drive Array (array b)

Surface Area Configuration Allow Remote Connections (Named Pipes and TCP/IP)