.NET StockTrader Technical Documentation

An End-to-End Sample Application Illustrating Windows Communication Foundation and .NET Enterprise Technologies

6/4/2007

Microsoft .net Framework

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

Contents

Introduction

The .NET StockTrader is an end-to-end sample application based on an online stock-trading scenario. The application is a new Microsoft .NET sample application designed to illustrate the use of the Windows Communication Foundation (WCF) technologies in an end-to-end service-oriented architecture. As such, the application illustrates many best-practice programming practices for .NET and WCF including the use of an n-tier, service-oriented design. As a benchmark sample and downloadable benchmark kit, the application also illustrates best-practice programming for building high-performance and scalable service-oriented applications using Microsoft .NET and WCF.

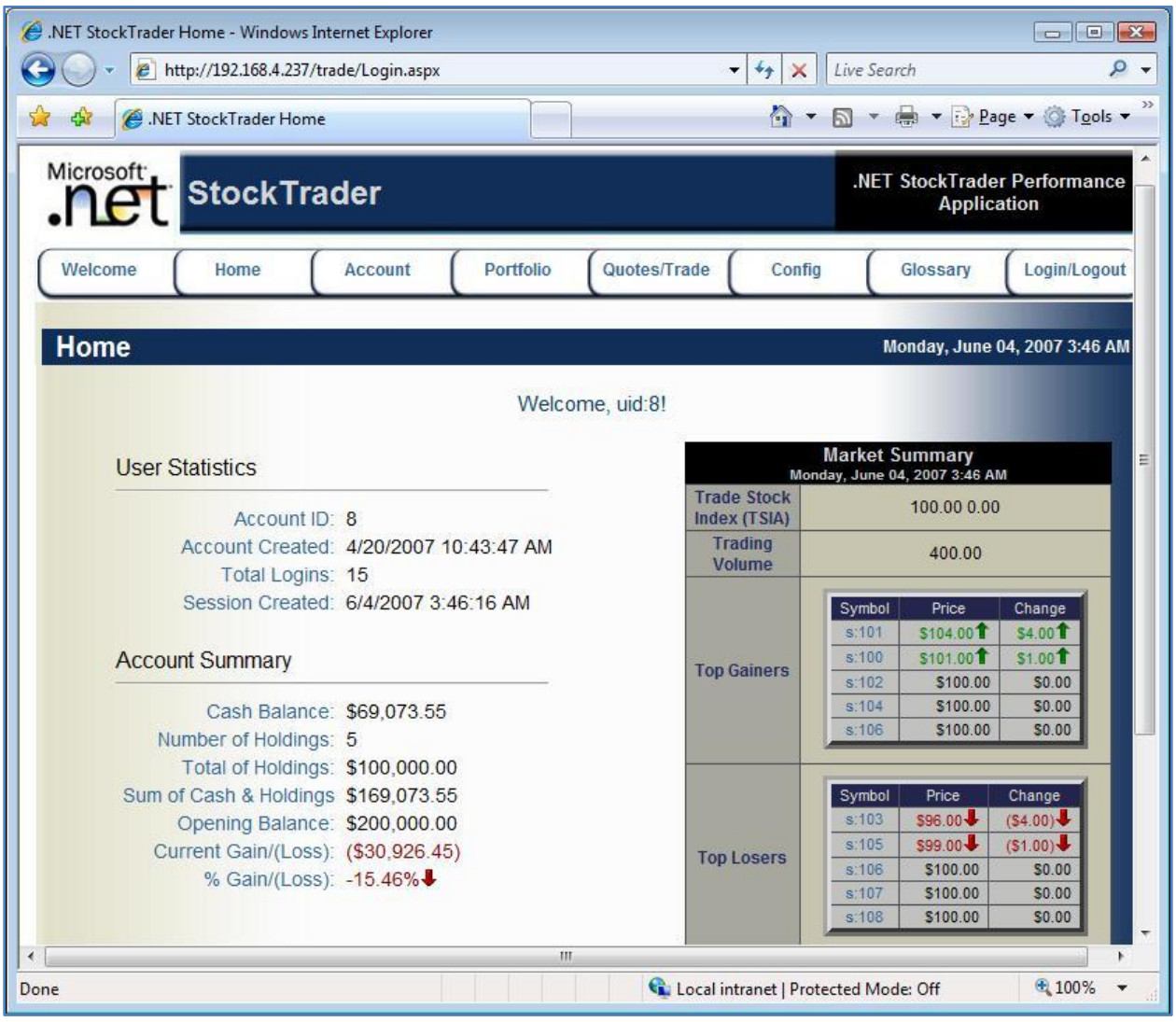

Figure 1: .NET StockTrader Home Page

.NET StockTrader and J2EE Interoperability

The application is functionally equivalent to the IBM WebSphere Trade 6.1 sample benchmark application, and serves to illustrate interoperability between .NET and IBM WebSphere. All of the

services in the application are exposed, via WCF, as industry-standard Web Services. The .NET StockTrader Web application seamlessly interoperates with the backend J2EE services of the WebSphere Trade 6.1 application. Conversely, the WebSphere Trade 6.1 Web application is able to seamlessly utilize the backend services of the .NET StockTrader. In each case, a simple endpoint configuration allows this interoperability, with no code changes required to either application.

Using .NET StockTrader to Compare to IBM WebSphere 6.1 and J2EE

The .NET StockTrader serves as a useful comparison between .NET best-practice architectures and programming approaches and IBM WebSphere best-practice architectures and programming approaches. Both WebSphere Trade 6.1 and the .NET StockTrader applications are designed as benchmark applications to illustrate the scalability and performance of their relative platforms. Since they are functionally equivalent with full interoperability, the .NET StockTrader application is also useful as a benchmark kit for comparing the scalability and performance of .NET/Windows Server-based applications to IBM WebSphere/J2EE-based applications across a variety of different alternative architectures. Microsoft has published .NET StockTrader vs. IBM WebSphere Trade 6.1 benchmark results based on standardized hardware configurations. The materials are available at http://msdn.microsoft.com/stocktrader.

Technologies Incorporated into .NET StockTrader

The following technologies are illustrated in the .NET StockTrader benchmark application:

- Interoperability between .NET and J2EE services based on WCF and industry-standard Web Services.
- Implementing high-performance ASP.NET Web applications with a logical n-tier, service-oriented enterprise design pattern.
- Implementing high performance WCF services.
- Implementing multiple service bindings to support different network transports and message encoding formats using WCF.
- Hosting WCF-based Web Services using IIS and self-hosting WCF Web Services within custom service hosts.
- Building loosely-coupled message-driven services utilizing WCF and MSMQ.
- Integrating with .NET 2.0 distributed transaction services by utilizing System.Transactions, the WCF transaction model and the Microsoft Distributed Transaction Coordinator
- Using WCF to implement systems with replicated messaging engines and transacted, durable messaging.
- Core performance tuning parameters for WCF and .NET to achieve high-throughput.
- Alternative physical deployment topologies inclusive of deploying to load-balanced server clusters for scalability and failover purposes.

.NET StockTrader Configuration Overview

There are four main components to the .NET StockTrader application, which is a service-oriented, composite application. These components include:

1. The .NET StockTrader Web Application user interface
2. The .NET StockTrader Business Services
3. The .NET StockTrader Order Processor Service
4. An optional Windows Presentation Foundation (WPF) smart client interface

The application can be configured to run the Web application and the two primary services (Business Services and Order Processing) within a single monolithic application (a single process hosts all three elements), or to remotely activate the Business Services and/or the Order Processor Service based on the use of Windows Communication Foundation. The WPF client always uses a remote interface to the middle tier services. The configuration is managed through a Configuration Management Service with backing SQL Server 2005 configuration repositories for each element of the application. While the .NET StockTrader uses the configuration management system and the source code for this system is included with the .NET StockTrader application, the configuration management system itself is separate from the application, and is designed to be re-usable across any application that might benefit from implementing it. This paper focuses on the core .NET StockTrader application; see the separate whitepaper entitled *Implementing Application Load Balancing and Centralized Configuration Management Repositories for .NET Applications and Services: .NET StockTrader Sample Application Scenario* for a more thorough technical overview of the configuration management service itself, beyond the shorter overview provided below.

Brief Overview of the .NET StockTrader Configuration Management Service

As a benchmark application, the .NET StockTrader is rich with different configuration settings to allow comparative performance testing of different designs, technologies, transport protocols, remoting options and the like. For example, the application can be configured to run services in-process, or to remotely access services using WCF. Once services are remoted, however, a best-practice design is to ensure they remain 'black-boxes' and completely autonomous from any applications or other services that may use them. Just because the StockTrader Business Services are used by the .NET StockTrader Web application, for example, does not mean they will not be used by other applications as well. Also, these remote services might be hosted at an application service provider, deployed in a different data center/geographic location within an Intranet, or even simply deployed behind a firewall on a separate subnet.

Each of these cases presents challenges in how to centrally manage and configure a distributed, composite application made up of different autonomous and remotely activated services. Such challenges are introduced based on the distributed nature of this application. Consider, for example, that even though IBM WebSphere supports clustering, to make changes to the configuration of Trade 6.1 in a cluster setup requires that each server be separately with the Trade 6.1 application-specific settings each time it is restarted. The .NET StockTrader configuration management implementation is

designed to overcome these challenges, and ensure the application not only performs well, but also can be more easily managed in a benchmark lab or production data center.

The .NET StockTrader uses two distinct remoteable services:

- The Business Services, which the Web Application calls into
- The Order Processing Service, which is called in turn by the Business Service layer (when configured for asynchronous order processing)

By abandoning assumptions about where services are located, where they are hosted or how they are hosted, greater flexibility is achieved in terms of supporting interoperability between services and different possible physical deployment topologies. So key to understanding .NET StockTrader is to understand that all three elements (Web Application, Business Services, Order Processor Service) are designed to be autonomous. The .NET StockTrader Web application uses and relies on Business Services, but has no awareness of the implementation details of the Business Services layer itself—such as its (optional) use of a separate Order Processing Service—or how it manages/uses its own configuration settings.

So configuring such a distributed system, where elements are running on different servers, possibly not even in the same data center, developed by different teams, possibly clustered across replicated servers, would be quite literally impossible without some way to centrally manage the configuration of the remote services. At the same time, we want to ensure each service remains autonomous; hence **each service must be responsible for its own configuration**. The IT Administrator, however, should have a way to 'view' and alter the overall composite application configuration in an integrated way. This is achieved via the Configuration Management Service, a re-usable system implemented in shared libraries and based on WCF for configuration exchanges between services and clustered nodes.

With .NET StockTrader, the configuration system is accessed and used simply by logging into the Web application as the pre-configured userid 'Admin'. This directs the user to the Configuration Menu, which is a set of ASP.NET pages that present a way to centrally view, manage and configure the overall system via the Configuration Service. These pages are generic: they could work with any application that implements the configuration management service—they are not necessarily StockTrader-specific.

Storing Configuration Data in a Service Configuration Repository

The core concept behind the .NET StockTrader configuration system is quite simple: instead of storing configuration parameters in configuration files (such as Web.Config or App/Exe.Config), all configuration information is moved into a SQL database: this is the service's configuration repository. Each service has its own configuration repository, and only that service itself is allowed to **directly** connect to/query/update its repository. So, for example, we should not assume the StockTrader Business Services have network connectivity to the Order Processor Service configuration repository database-it may be on a private/protected subnet, for example; or separated by a firewall, etc. Hence, Business Services must rely on the Order Processor Service itself to make changes to its own configuration database, vs. attempting to make direct updates from one service to another service's repository. Instead, such updates can be requested and "pushed" through the various service layers via the

Configuration Service and the StockTrader Configuration Menu. The Configuration Service is itself a WCF service that defines a contract (interface) implemented by each element of the application. However, the bulk of the logic for the system is contained in helper assemblies/pre-built shared libraries, to make it easier to implement the configuration service in other applications.

Services 'bootstrap" themselves on startup simply by loading their configuration settings from their configuration repository. The only element stored in the service's local config file is hence a pointer to its configuration repository. This means the application can simply be copied between servers to setup new clustered servers—with no manual updating of config files for each peer server required. The configuration settings are auto-loaded via .NET ***reflection*** into application-specific members (variables) that are treated like C# constants (they are actually static variables). Once loaded, each service host has its centrally managed configuration and is ready to service requests. Any application-specific setting could be stored in the repository—for example, the default timeout on transactions; the length of edit windows in a Web application, user exception messages, the Event Log name to log exceptions/messages to; tuning parameters; connection URIs to other remote services, etc. Because these settings are no longer "hard-coded" into the application, making global configuration changes to a running service or application requires no recompile, redeploy or even re-start of the running service instance.

So a core concept behind StockTrader is to **enable much easier management and ongoing data center operations for the service-oriented application.** For example, by allowing dynamic updates to the configuration, many (if not most) application recompiles and re-deploys can be avoided if they do not involve core changes to the business logic itself. This means avoiding the need to stop/start production services/applications, redeploying changes, and the associated downtime or management costs to make configuration changes to a running, clustered service/application.

Once such configuration data for a service or application is stored centrally, it's easy to see how services that are horizontally clustered can be much more easily configured and managed. The Configuration System, for example, automatically updates running peer nodes with changed configuration data, keeping servers constantly in a synchronized state. All three elements of the .NET StockTrader Application (Web Application, Business Services, Order Processor Service) implement this configuration system.

Dynamic Clustering

One idea incorporated into the sample application is the idea of ***virtualizing*** services across any number of clustered nodes. This is also a core concept implemented in the Configuration Management System. Instead of manually/statically setting up and configuring clusters as with most application servers, .NET StockTrader services are automatically clustered/load balanced simply by starting them up on different servers with a pointer (connection string) to a common repository. This is possible because each service node points to the same central configuration repository—which is also its cluster management system. Hence, Business Services, for example, is 'clustered' for load balancing/scalability and application level failover rather easily: simply start the service on two different machines.

Connection Points

In the sample application, services are connected via the concept of *connection points.* Connection points are simply addresses (URIs) to remote services. Once an administrator of the StockTrader application configures a connection point (via the Connection tab in the Configuration Menu) to a running instance of a remote service, the configuration system automatically maintains and manages connections to the entire 'virtualized' service; such that load balancing and failover happen automatically. This is not so different than traditional application servers such as WebSphere, where client proxies (such as the IBM HTTP Server plug-in for WebSphere) maintain lists of backend servers and perform load balancing/failure detection against them. However, as new service instances within a virtualized service cluster are brought online, they will start receiving load and participate in the service cluster automatically. As they are closed, requests from clients will not be directed to these remote service instances. This automatic virtualization can be utilized by any application/service implementing the Configuration Management System. The load balancing is simple round-robin, but since the source code is included with the sample, customers could alter the code and implement other algorithms.

Connection points to services that do not implement the Configuration Management system can also be established: for example, connecting the .NET StockTrader Web Application to the IBM Trade 6.1/J2EE services is as simple as adding a connection point to a remote IBM WebSphere server running Trade 6.1.

Health Monitoring

The ASP.NET configuration menu pages within the StockTrader sample application allows the administrator to view the various servers participating in a cluster, and see which servers are online or offline.

.NET StockTrader Application Design

The core .NET StockTrader design is based on an enterprise, n-tier design pattern with full logical separation of the application into three distinct layers:

1. Web Application Layer (UI), utilizing ASP.NET and Web Forms
2. Middle layer Business Services Layer (BSL), utilizing stateless C# classes
3. Data Access Layer (DAL), utilizing ADO.NET and stateless C# classes

In addition, C# classes are used to model data records as mapped from the RDBMS database tables. The model classes are passed between all layers in the application allowing complete separation of the database implementation from both the business services and user interface layers.[1]

[1] Model classes are also used in the WebSphere Trade 6.1 application, and the .NET StockTrader application uses the same model class definitions on the service tier such that full Web Service interoperability can be achieved between the two platforms. The backend service model classes are automatically mapped into WSDL by .NET, and serialized as XML for seamless J2EE interoperability.

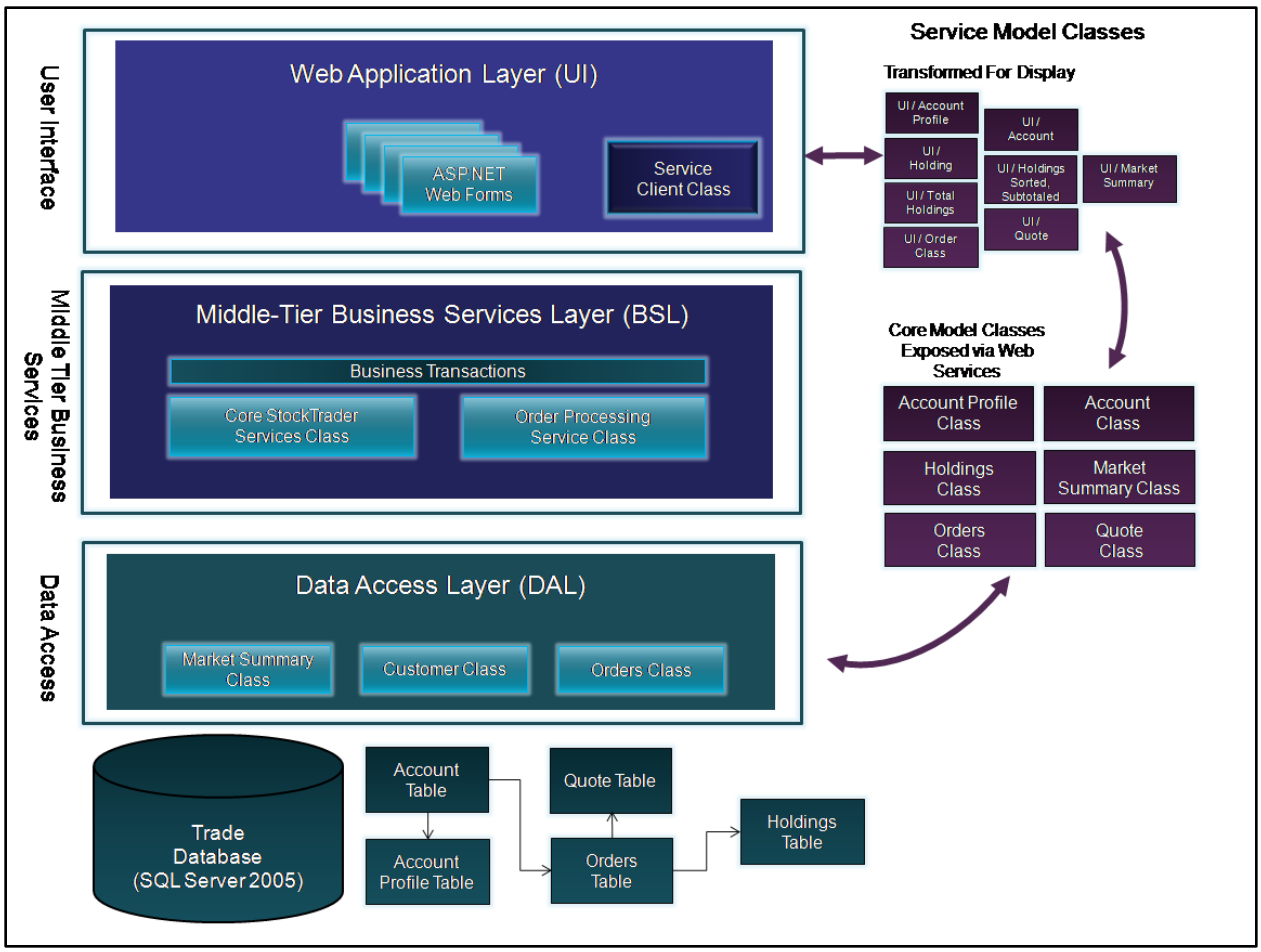

Figure 2: .NET StockTrader Logical n-tier Design

Based on this logical layering, several advantages are realized:

- The implementation logic of the business services can be changed without making any changes to the other layers.
- Database-specific implementation details are hidden to the Web UI Layer and the Business Services Layer. Hence, additional data access assemblies (DAL classes) can easily be added to communicate with other types of databases (e.g. DB2, Oracle). This can be done without making any changes to the Web/UI layer or the Business Services Layer.
- Services can be co-located or remotely invoked by the Web layer; enabling more flexible deployment options. For example, The ASP.NET Web servers do not need to have any database connectivity or access, since they only invoke methods in the Business Services Layer.
- The design is horizontally scalable out of the box across load-balanced server clusters using the built-in StockTrader Configuration System[2], Windows Server Network Load Balancing or hardware load-balancing techniques. All user state is cluster safe without the need to modify IIS

[2] See the separate paper *Implementing Application Load Balancing and Centralized Configuration Management Repositories for .NET Applications and Services: .NET StockTrader Sample Application Scenario*.

session state configurations. Both the Web UI Layer and the Business Services Layer can be easily clustered and also deployed on separate clusters on opposite sides of a firewall.

- Different facades can be used to front-end the Business Services Layer; as discussed later in this paper, the .NET StockTrader has both ASMX and WCF Web Service facades to the Business Services Layer.[3]
- Different programmers can more easily work as team, each focusing on a specific layer.

.NET StockTrader Configuration Options

The application can be easily configured to use different .NET technologies which are incorporated into the design. This allows for technology comparisons across programming model and performance, to aide architects in making architectural decisions for their own applications. The alternative configurations also illustrate the flexibility of WCF and .NET to support different physical deployment topologies, network transports, and message-encoding standards. The .NET StockTrader application has two primary settings: **AccessMode** and **OrderMode**. AccessMode determines how the ASP.NET StockTrader Web application invokes the backend services. OrderMode determines whether orders are processed in-process/synchronously or remotely/asynchronously, and over which messaging format and specific transport protocol.

.NET StockTrader AccessMode Settings

Setting	Description
InProcess	The Web application invokes the business service layer via a direct, in-memory mode with no remote service calls. In this setting, the Web application and business service layer run as a single, monolithic ASP.NET Web application and the Web Layer and Business Service Layer cannot be divided across a network.
Asmx_WebService	The Web application invokes the service layer via ASMX Web Services using SOAP; Http and Text-XML encoding. In this setting, the Web layer and Service layer can be divided across a network, and any Web Service client (.NET or J2EE) can utilize the backend StockTrader Web Services.
IISHost_WebService	The Web application invokes the service layer via WCF Web Services hosted in IIS. It uses SOAP; Http and Text-XML encoding, and is the WCF equivalent of ASMX Web Services (which are always hosted within IIS). In this setting, the Web layer and Service layer can be distributed across a network, and any Web Service client (.NET or J2EE) can utilize the backend StockTrader WCF services.
Http_WebService	The Web application invokes the service layer via WCF Web Services self-hosted in a Windows application (not IIS). It uses SOAP; Http and Text-XML encoding, and is an example of self-hosting Web Services with WCF. In this setting, the Web layer and Service layer can be distributed across a network, and any Web Service client (.NET or J2EE) can utilize the backend StockTrader WCF services.

[3] The ASMX façade is included only for benchmark comparative purposes, since WCF provides a true superset of features over ASMX and replaces ASMX.

Tcp_WebService	The Web application invokes the service layer via WCF Web Services hosted in a Windows application (not within IIS). It uses the TCP/IP transport and binary encoding, and is an example of self-hosting Web Services with WCF. This mode is roughly analogous to using .NET 2.0 binary remoting; however, it uses the WCF service programming model which replaces .NET binary remoting by removing the programming distinction and presenting a single model for building services, no matter the remoting transport or encoding format used. In this setting, the Web Layer and Service Layer can be distributed across a network, and any .NET client can utilize the backend StockTrader WCF services over the binary format, while simultaneously providing the ability for non-.NET clients to fully interoperate over SOAP, Http and text-xml encoding.

.NET StockTrader OrderMode Settings

Setting	Description
Sync_InProcess	Orders are processed synchronously by the order processing service, and the Order Processing Service is run in-process with Business Services---so not remote calls are made for order processing.
Async_Msmq	Orders are placed asynchronously via WCF with an MSMQ binding. The WCF service is fully transacted, with orders placed into a transacted, durable message queue with assured message delivery. The StockTrader order placement service invokes a separate order processing service via WCF. In this mode, the order processing service can be distributed across a network and run on a separate, dedicated application server/server cluster from the main BSL layer. This is an example of loose coupling, since the order processing service does not have to be online for users to actually place trade orders (stock buys and sells). When the order processing service is brought online, it will automatically process any orders in the queue. Additionally, since the order processing service is based on WCF, the developer does not program any MSMQ-specific logic; rather they simply program to a standard WCF service, and use an MSMQ binding for that service—WCF takes care of all the rest.
Async_Msmq_Volatile	Orders are placed asynchronously via WCF with an MSMQ binding. The WCF service is bound to an in-memory, non-transacted MSMQ and hence not persisted to disk. This mode enables performance comparisons with the ASync_Msmq setting above to compare performance differences between using durable vs. non-durable messaging. This is also an example of loose coupling, since the order processing service does not have to be online for users to place trade orders (stock buys and sells). When the order processing service is brought online, it will automatically process any orders in the volatile (in-memory) MSMQ queue.
Async_Tcp	Orders are placed asynchronously via WCF with a TCPIP transport binding. Order processing is asynchronous and the order processing service can be distributed across the network and run on a separate, dedicated application server/cluster. However, because MSMQ is not

	being utilized, this is a tightly coupled design, in that the order processing service must be online for users to place trade orders.
Async_Http	Orders are placed asynchronously via WCF with an Http transport binding. Order processing is asynchronous and the order processing service can be distributed across the network and run on a separate, dedicated application server/cluster. However, because MSMQ is not being utilized, this is a tightly coupled design, in that the order processing service must be online for users to place trade orders.

Any combination of AccessMode and OrderMode settings is viable with the .NET StockTrader application.

Useful Benchmark Comparisons

The StockTrader application presents an end-to-end test bed for benchmarking various technologies that make up WCF and the .NET platform. The following table illustrates some of the comparisons that can be made for the scenario using the downloadable StockTrader benchmark kit:

- In-memory activation performance vs. Web Service performance
- WCF vs. ASMX Web Service performance
- IIS-hosted WCF services vs. self-hosted WCF services
- Web Services with an Http binding and text-xml message encoding vs. a TCPIP binding with binary message encoding
- Asynchronous WCF messaging over a durable MSMQ vs. non-durable WCF messaging options
- .NET performance vs. IBM WebSphere/J2EE EJB entity bean performance
- .NET performance vs. IBM WebSphere/JDBC data access performance
- .NET WCF Web service performance vs. IBM WebSphere Web Service performance
- WCF with MSMQ messaging vs. IBM WebSphere Enterprise Service Bus (which is based on JMS/WebSphere Service Integration Bus (SIB) messaging)

These comparisons and others are published and discussed at length in the paper *Benchmarking .NET StockTrader vs. IBM WebSphere Trade 6.1.*

.NET StockTrader Access Mode Configuration Details

The following sections discuss each of the configuration options for .NET StockTrader Web Application, including architecture illustrations, in greater detail. The settings discussed are configured simply by changing the AccessMode setting for the StockTrader Web application using the configuration menu. Additionally, for remote modes, a *connection point* will need to be initially established to the remote (running) service via the Connections Tab in the configuration menu. Please refer to the *.NET StockTrader Installation and Configuration* paper for details. Multiple connection points can be established and saved (for example, to WebSphere, to IIS/ASMX services, to IIS/WCF services, to self-host/WCF services)—the ones that are used are determined by the current AccessMode setting.

In-process invocation of the backend services

This base design represents a standard enterprise design for building scalable Web applications that do not require remote service invocation or a loosely-coupled message-oriented architecture. To run the application in this mode, set the AccessMode setting for the Web application to InProcess. When running in this mode, the application can be illustrated as follows[4]:

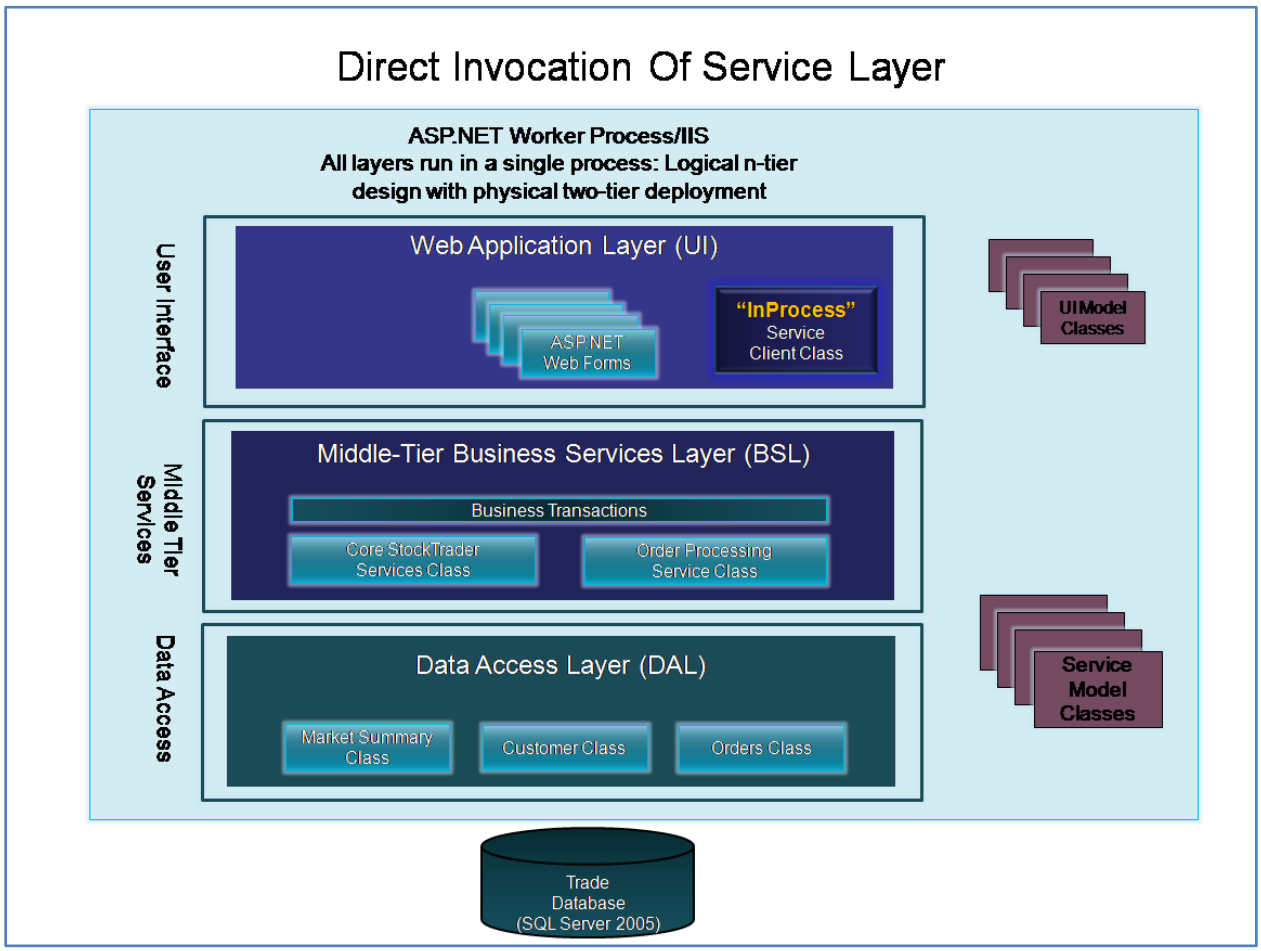

Figure 3: .NET StockTrader with InProcess Access Mode

In this mode, all layers of the application are co-located on an (or cluster of replicated) application server(s) running Windows Server 2003 with Internet Information Server (IIS), and .NET 2.0. The application is hosted in a dedicated Common Language Runtime service host running as an ASP.NET worker process in conjunction with the IIS Web server. The database runs on a dedicated, remote SQL Server 2005 database server. This design presents a very fast and horizontally scalable application that can be deployed on either a single application server or across multiple, load-balanced. Such physical deployment topologies are shown below:

[4] Note that this mode precisely corresponds to running WebSphere Trade 6.1 in the Standard Access Mode as opposed to the Trade 6.1 Web Service Access Mode. All calls in Trade 6.1 between the JSPs and EJBs are local calls, and they run in the same JVM process. Please refer to the *Benchmarking .NET StockTrader vs. IBM WebSphere Trade 6.1* benchmark paper for more details.

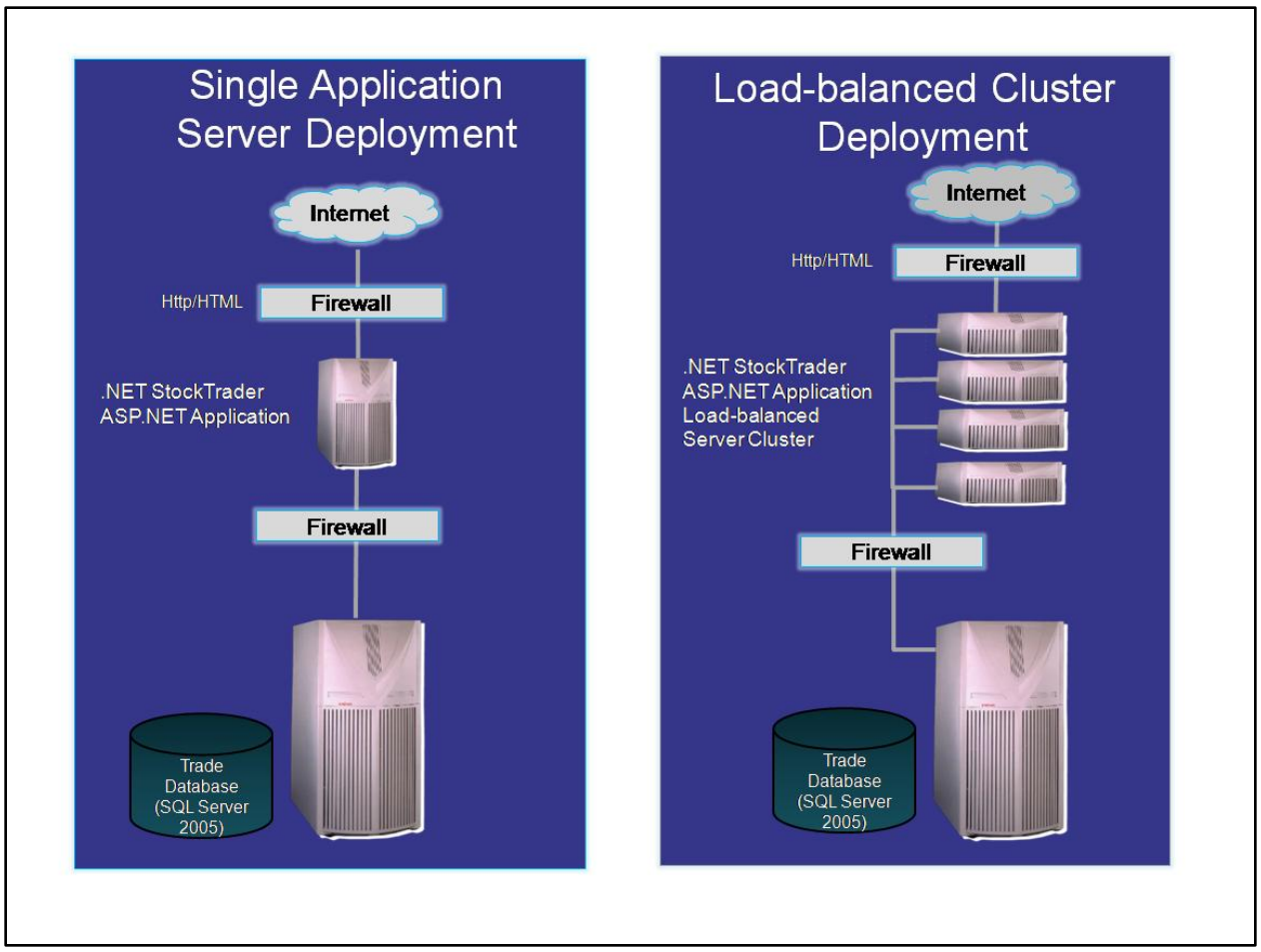

Figure 4: Deploying StockTrader in InProcess Mode

Design Considerations

The characteristics of such a design are:

- Fast, in-process activation of all layers of the application
- A single application package can be easily deployed/replicated across machines
- Vertical scaling via additional processors
- Horizontal scaling via load-balanced clusters of replicated servers
- Failover/high availability via load-balanced clusters of replicated servers

However, back-end services are only accessible via a single, .NET-based application – the ASP.NET Web application. Other .NET applications and non-.NET applications cannot utilize these services. In addition, the workload of the application cannot be separated: all servers execute both the ASP.NET Web/layer and the backend services layer.

Remote invocation of backend services hosted within IIS

This design represents a standard enterprise design for building scalable Web applications that require remote invocation of back-end services and physical separation of the front-end Web application and

the backend processing including data access logic. The .NET StockTrader can be configured for this design simply by changing the AccessMode configuration setting to:

- Asmx_WebService: backend services are remotely activated via ASMX Web Services over Http with text-xml encoding and the System.Xml.Serialization framework classes. The backend services, as with all ASMX Web services, are hosted within IIS.
- IISHost_WebService: backend services are activated over WCF Web Services over Http with text-xml encoding and the WCF DataContractSerializer (.NET 3.0 System.Runtime.Serialization). The backend services are hosted within IIS.

When running in this mode, the application architecture and deployment options can be illustrated as follows[5]:

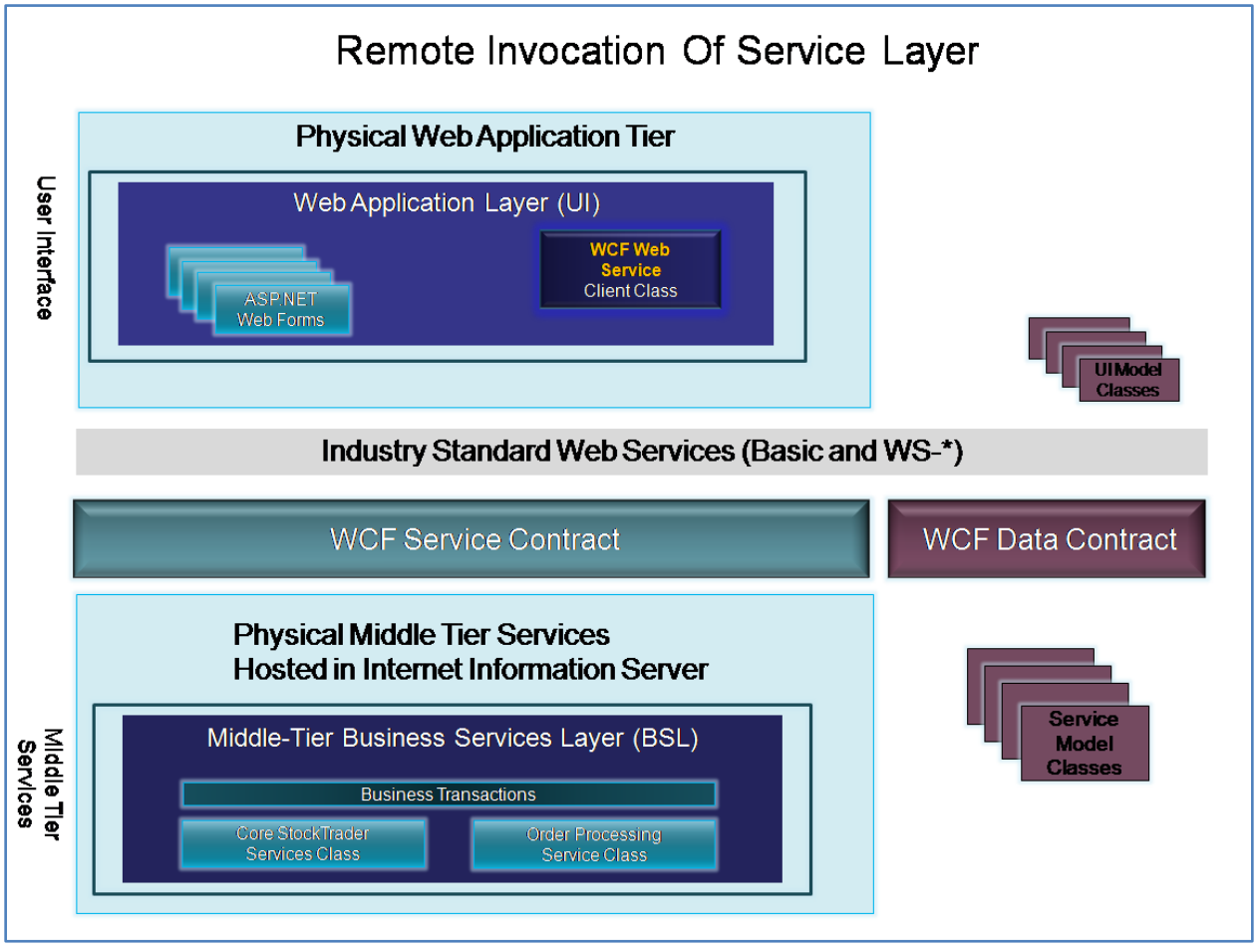

Figure 5: .NET StockTrader in Asmx_WebService or IISHost_WebService Access Modes (Data Access Layer not depicted)

[5] Note that these Web service modes equate to running WebSphere Trade 6.1 in the Web Service interface mode. Please refer to the *Benchmarking .NET StockTrader vs. IBM WebSphere Trade 6.1* benchmark paper for details.

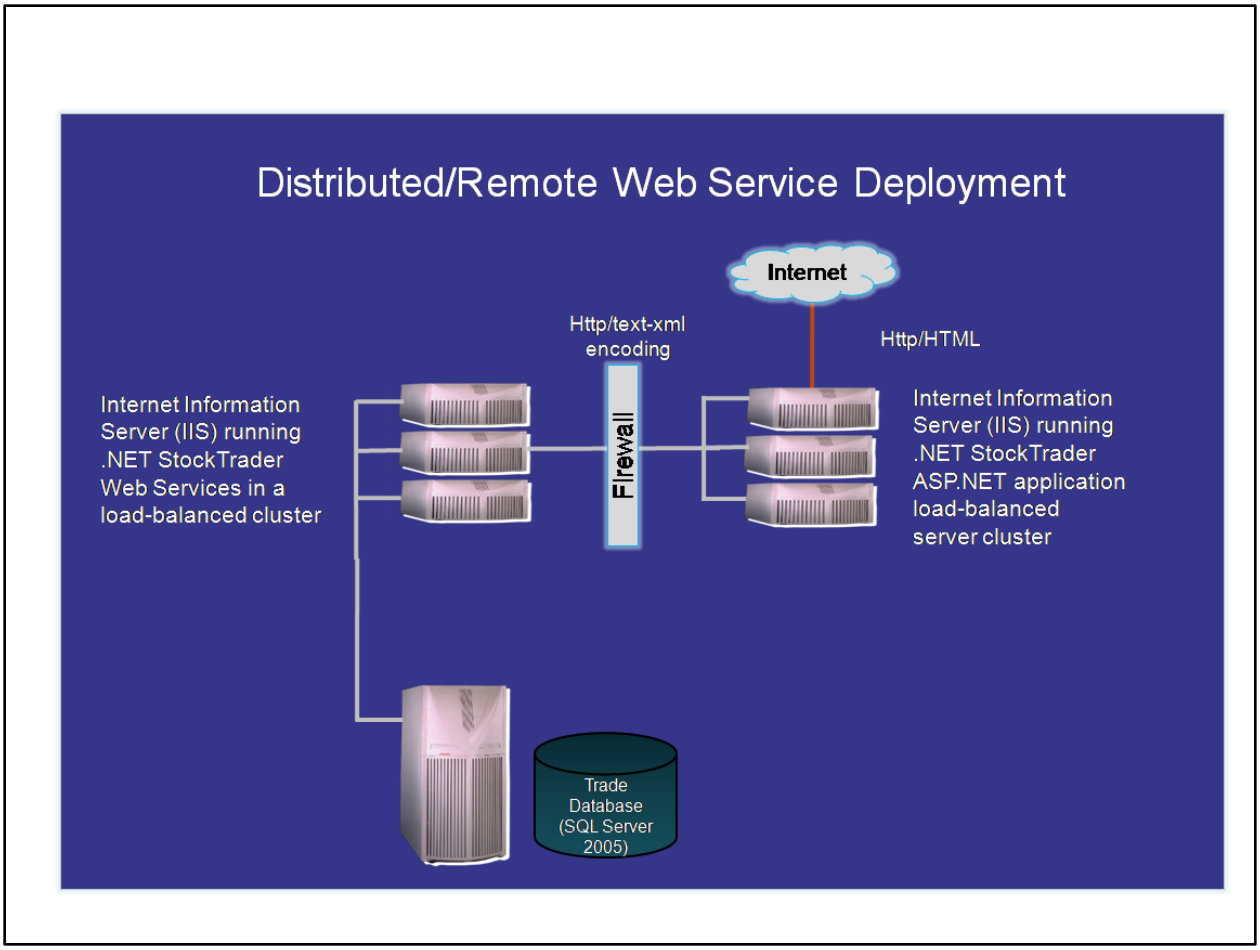

Figure 6: .NET StockTrader in Remote Web Service Deployment with IIS-hosted services

In this mode, the ASP.NET StockTrader application is physically partitioned from the backend business services and their corresponding data access operations. The ASP.NET Web layer runs on a single (or cluster of replicated) application server(s) running Windows Server 2003 with Internet Information Server (IIS), and .NET 2.0. The application is hosted in a dedicated Common Language Runtime service host, running as an ASP.NET worker process in conjunction with the IIS Web server. The service layer, in turn, runs on a separate application server (or server-cluster) also running IIS and .NET to host the remote Web services. These services can hence be consumed by other applications, including non-.NET applications (such as J2EE applications) since ASMX and WCF Web Services are based on the industry standard SOAP protocol running over the Http transport with XML encoding. As with the first design, the database runs on a dedicated, remote SQL Server 2005 database server. This design presents a horizontally scalable service-oriented application. The Web layer and backend service layer can each be hosted on their own dedicated application servers or across separate dedicated application server clusters on different subnets.

Design Considerations

Characteristics of such a design are:

- Potential to separate UI layer from backend services for operation maintenance/support by separate teams (potentially geographically disperse)
- Potential to build multiple user applications that use the common .NET StockTrader Web Services via service-orientation design (service re-use)
- Ability to provide additional scale by partitioning Web front-end workload from the backend services/data access workload; spreading the workload across dedicated servers
- Vertical scaling via additional processors for all layers of application
- Horizontal scaling via load-balanced clusters of replicated servers (applies to both physical layers)
- Failover/high availability via balanced clusters of replicated servers
- Remote activation of services via Http (internet or intranet remoting)
- Interoperability with Java/J2EE applications for both ASP.NET Web application (accessing backend J2EE services via ASMX client) and backend .NET ASMX Web services (now accessible by other J2EE/Java-based front-end applications)

Remote invocation of self-hosted WCF Web Services

This design is very similar to the previous Web Service design, except it utilizes the ability to self-host Windows Communication Foundation services vs. hosting all Web Services within ASP.NET/IIS worker processes. WCF, shipped as a core component of .NET 3.0, introduces a more sophisticated set of features for SOA vs. ASMX/.NET 2.0, and a much more flexible set of transport, encoding and hosting options. It also introduces support for the latest WS-* Web Service standards such as WS-Reliable messaging, WS-Security and WS-Transactions. As such, it consolidates the MS Web Services Enhancement toolkit (WSE), ASMX, and .NET remoting into a single, integrated service programming model and runtime. At the same time, .NET 3.0 is an extension of the .NET 2.0, and does not introduce a new Common Language Runtime or replace other elements of .NET 2.0, so adding WCF features to an application does not require redeployment or versioning of the core .NET 2.0 framework or runtime.

To run the application in this mode, set the AccessMode setting to:

- Http_WebService: backend services are remotely activated via WCF Web Services over Http with text-xml encoding and the WCF DataContractSerializer. The backend services are hosted within the WCF TradeWebServiceHost.exe console application.
- Tcp_WebService: backend services are activated over WCF Web Services over TCP with binary encoding and the WCF DataContractSerializer. The backend services are hosted within the WCF TradeWebServiceHost.exe console application.

In this mode, as in the previous IIS-hosted service modes, IIS and the ASP.NET application reside on one physical server (or load-balanced) server cluster, while the middle-layer business services and data access services reside on a separate application server (or load-balanced application server cluster). Several new and more flexible deployment options are introduced for this design based on new WCF features. These include the ability to easily configure transport protocols for the remote service calls, and the ability to build self-hosted services that do not run or require an IIS Web server installation for the backend service layer. In addition, services can simultaneously respond over multiple endpoints,

each with unique transport protocols and configuration options. WCF fully supports the WS-* Web services standards, so it also offers full interoperability with SOAP-based services running on non-Microsoft platforms, such as J2EE. However, unlike ASMX, a single codebase/programming model allows developers to utilize faster; binary remoting mechanisms for .NET-based clients, while simultaneously offering full XML/SOAP/Http interoperability with connecting clients from other platforms such as J2EE. When running in this mode, the application can be illustrated as follows[6]:

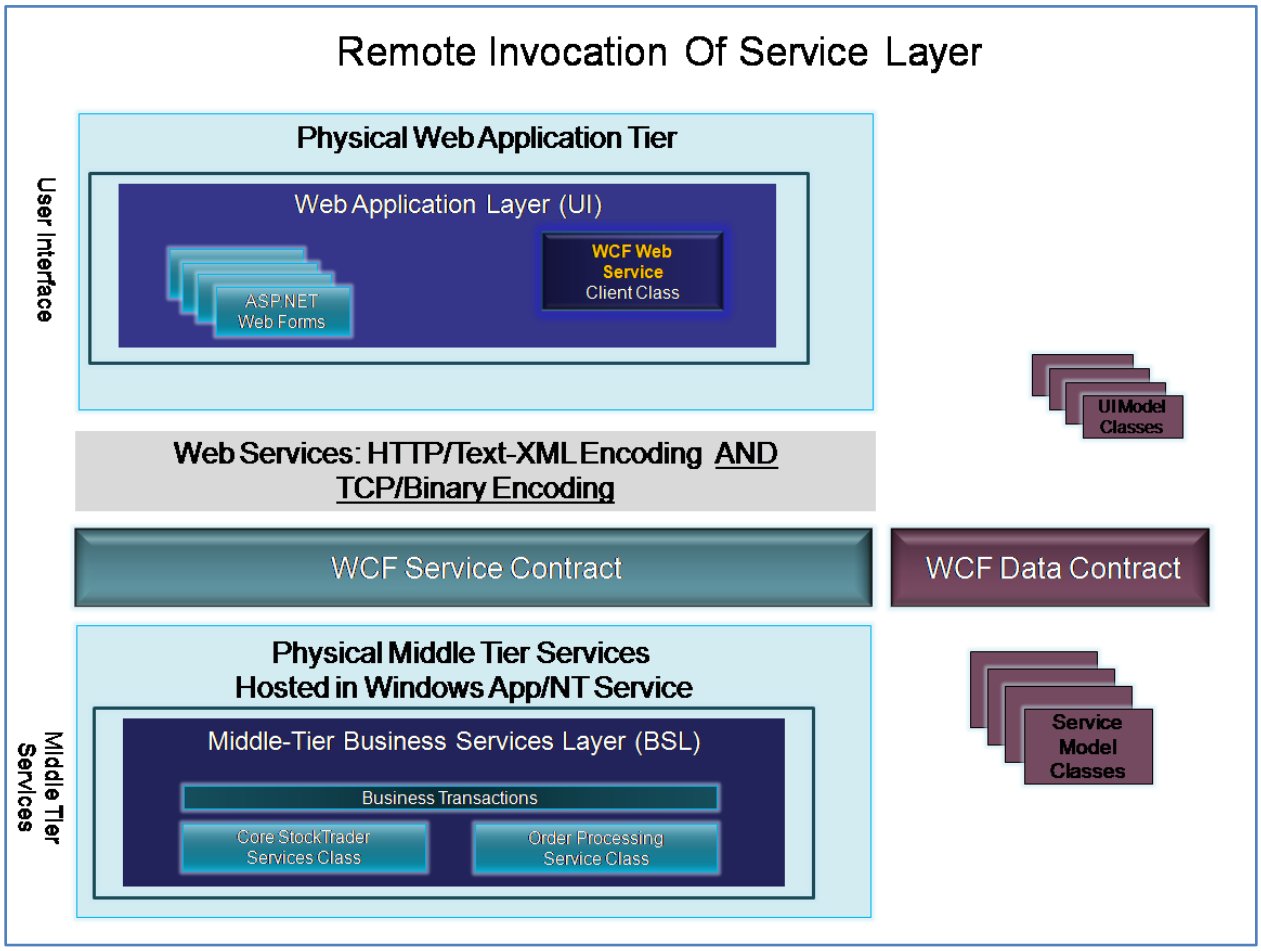

Figure 7: .NET StockTrader in Http_WebService or Tcp_WebService Access Modes

[6] Note that these Web service modes equate to running WebSphere Trade 6.1 with the Web Service Access Mode setting.

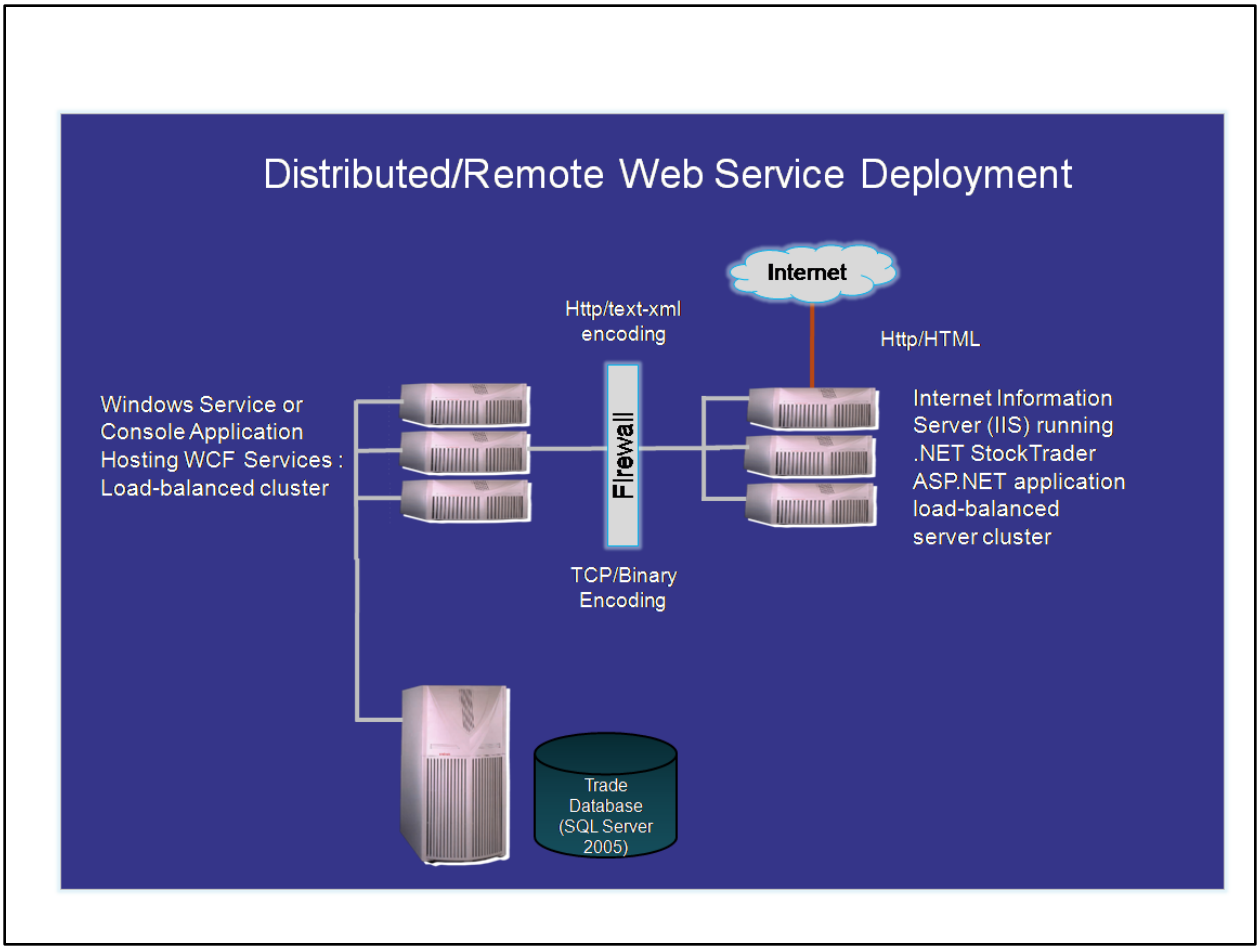

Figure 8: .NET StockTrader in Remote Web Service Deployment with self-hosted WCF Web Services

In this mode, the ASP.NET StockTrader application is physically partitioned from the backend business services and their corresponding data access operations. The ASP.NET Web layer runs on a single (or cluster of replicated) application server(s) running Windows Server 2003 with Internet Information Server (IIS), and .NET 2.0. The Web application is hosted in a dedicated Common Language Runtime service host, running as an ASP.NET worker process in conjunction with the IIS Web server.

The service layer, in turn, runs on a separate application server (or server-cluster). This server is running a dedicated process (NT Service, Console application or Windows application) that is self-hosting the WCF services. In this mode, .NET clients can optionally connect via binary encoding over TCP (vs. Http), which can offer enhanced performance. This comes without sacrificing interoperability with non-MS platforms, *since the WCF service host is simultaneously supporting Http/text-XML and Tcp/binary endpoints*.

As with the first design, the database runs on a dedicated, remote SQL Server 2005 database server. This design presents a horizontally scalable service-oriented application. The Web layer and backend service layer can each be hosted on their own dedicated application servers or across separate dedicated application server clusters.

The .NET StockTrader Self-Host Executable

While WCF supports self-hosting services in .NET console applications, Windows Services, or even Windows applications, the current implementation of .NET StockTrader hosts the services within a .NET Windows application. This allows for interactive input and optional display of on-screen activity messages. This application is named "Trade.BusinessServiceHost.exe." Please refer to the *.NET StockTrader Installation and Configuration* paper for details on running this .NET program on Windows Server.

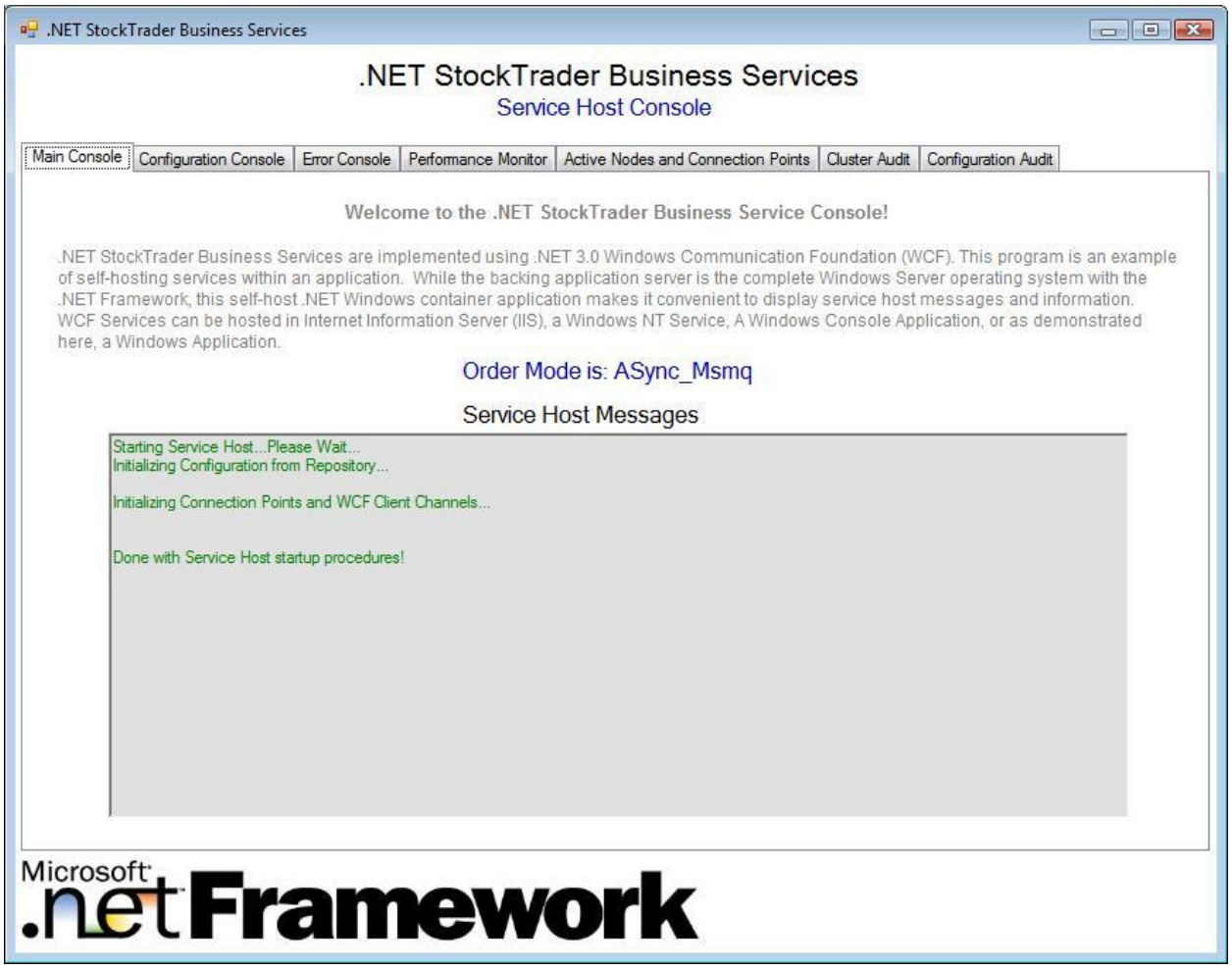

Figure 9: WCF Self Host Program for .NET StockTrader Business Services

Design Considerations

Characteristics of such a design are:

- Potential to separate UI layer from backend services for operation maintenance/support by separate teams (potentially geographically disperse)
- Potential to build multiple user applications that use the common WCF .NET StockTrader Web Services via service-orientation
- Can provide additional scale by partitioning Web front-end workload from the backend services/data access workload; spreading the workload across multiple machines

- Vertical scaling via additional processors for all layers of application
- Horizontal scaling via load-balanced clusters of replicated servers (applies to both physical layers)
- Failover/high availability via balanced clusters of replicated servers
- Remote activation of services via Http (internet or intranet remoting)
- Interoperability with Java/J2EE applications for both ASP.NET Web application (accessing backend J2EE services via a WCF client) and backend .NET WCF Web services (now accessible by other J2EE/Java-based front-end applications)
- Ability to have clients that can connect over multiple protocols including Http, TCP and others (simultaneously)
- Ability to simultaneously support XML/Text encoding and .NET Binary encoding
- .NET binary encoding and TCP transport offer increased performance
- Ability to self-host backend Trade Services vs. hosting only in IIS
- Support for latest WS-* standards such as WS-Security, WS-Reliable Messaging and WS-Atomic Transactions
- Flexible, easier WCF programming model for building SOA applications

.NET StockTrader Order Mode Configuration Details

The second configurable setting for the application is the OrderMode setting, which determines how orders are processed on the backend. While AccessMode is a StockTrader Web Application Setting, OrderMode is a setting maintained by the Business Service layer. Hence, in the configuration menu, you will need to drill down to the Business Service configuration settings to change the OrderMode. Likewise, you will need to drill down into Business Services connection points (via the connection tab) to establish connections between Business Services and the Order Processing Service.

When OrderMode is set to Sync_InProcess, orders are processed synchronously and do not invoke a separate WCF service. However, like Trade 6.1, the application can be configured to utilize a separate service to asynchronously process orders. This WCF service runs within a separate self-hosted console application ("Trade.OrderProcessorHost.exe"), and can be physically separated (remotely invoked) from the middle tier service layer. As with the WCF self-hosted Web Services, the order processor service supports several different binding protocols and activation modes. These are easily configured via Web.Config and/or TradeWebServiceHost.exe.config. Please refer to the *.NET StockTrader Installation and Configuration* paper for details on running this .NET order processor self-hosted service on Windows Server.

Note that order *placement* is always synchronous since the Web UI expects a response from the order placement service as the user waits for confirmation of the order with a returned order number. However, the actual order processing (which involves more extensive business logic and numerous queries and updates to various database tables) can be configured as either synchronous or asynchronous. When set to an asynchronous setting as described below, the order placement service makes an asynchronous call to the separate order processing service, vs. processing the order at once on its own before returning a response to the client application.

Transaction Management for Order Placement

Even though orders might be processed synchronously or asynchronously, transactional processing of orders is *always* required as the order processing involves many different database operations that must be treated as an atomic unit of work. The .NET StockTrader application uses either (based on a user-setting managed in the repository) ADO.NET transactions or *System.Transactions* transaction processing capabilities within the .NET Framework 2.0 to accomplish this work. System.Transactions represents a newer and faster method of invoking transactions vs. COM+ Serviced Components. Instead of inheriting from the IServicedComponent interface and requiring COM+ interoperability overhead, System.Transactions interacts with the Microsoft Distributed Transaction Coordinator directly without this overhead. In addition, System.Transactions will not promote transactions to distributed transactions when operating on the same database connection on the same database. It will instead use 'lightweight' transactions, offering a potentially significant performance boost by not invoking two-phase commit/distributed transaction logging. However, should the programming logic invoke resources to a second transacted resource, it will automatically promote that transaction to a distributed transaction, shielding the developer from the implementation details. System.Transactions will work with all major database systems, but today supports lightweight transactions only on SQL

Server 2005. As with COM+ Serviced Components, System.Transactions infrastructure handles all logging and transaction coordination so the developer does not have to. For distributed transactions, since System.Transactions works through the DTC, it is fully XA compatible for operations against non-Microsoft database engines such as Oracle and DB2. The following code snippet illustrates the use of System.Transactions within the BSL for a buy order placement:

```
public OrderDataBean buy(string userID, string symbol, double quantity, int orderProcessingMode)
        {
            System.Transactions.TransactionOptions TxOps = new TransactionOptions();
            TxOps.IsolationLevel = System.Transactions.IsolationLevel.ReadCommitted;
            using (TransactionScope scope = new TransactionScope(TransactionScopeOption.Required,TxOps))
            {
                try
                {
                    IOrder dal = Trade.DALFactory.Order.Create();
                    OrderDataBean order = dal.buy(userID, symbol, quantity);
                    scope.Complete();
                    return order;
                }
                catch (Exception e)
                {
                    Util.LogError(e.Message, false);
                    throw new Exception(e.Message);
                }
            }
        }
```

Synchronous Order Processing

To run in synchronous order processing mode, the OrderMode setting is set to 'Sync_InProcess' within the Business Service configuration repository (via the config menu in the Web app). In this mode, no service-to-service call is made: the order placement service directly (and synchronously) processes each buy or sell order as requested by user input from the Web UI Layer. This mode is depicted below[7]:

[7] This mode directly corresponds to the WebSphere Trade 6.1 Synchronous Order Processing configuration. In this mode, Trade 6.1 places orders directly with no asynchronous interface for order processing. Please refer to the *Benchmarking .NET StockTrader vs. IBM WebSphere Trade 6.1* benchmark paper for details.

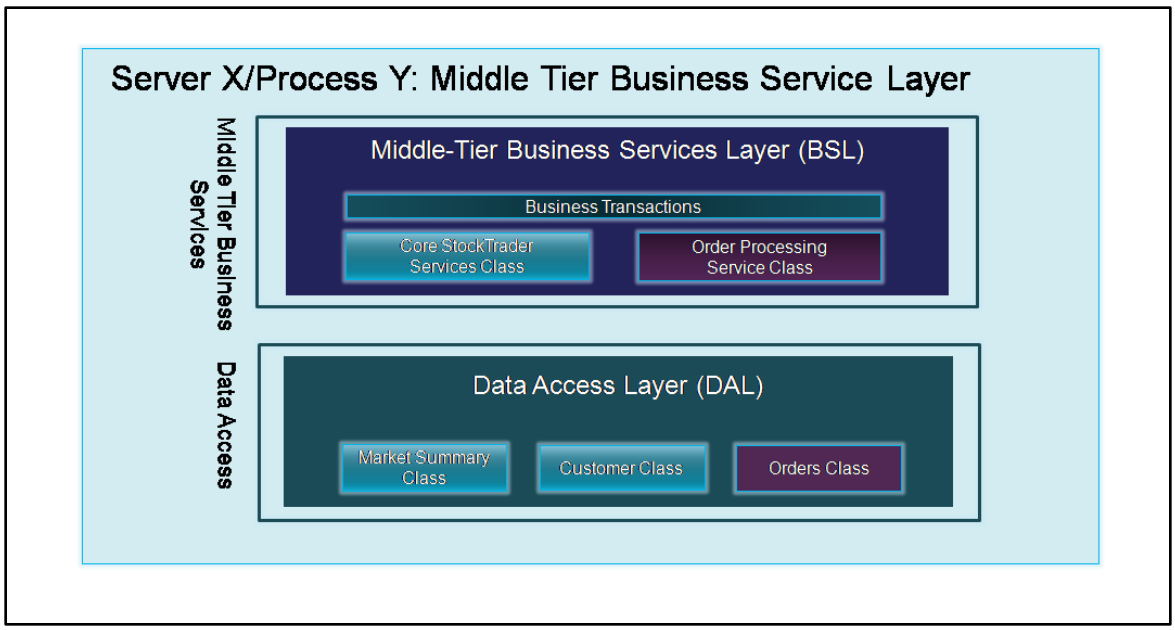

Figure 10: Synchronous Order Placement. User interface layer not depicted.

Design Considerations

The characteristics of such a design are:

- Fast, in-process activation of order processing
- Vertical scaling via additional processors
- Horizontal scaling via load-balanced clusters of replicated servers running the middle tier services
- Failover/high availability via load-balanced clusters of replicated servers
- No workload partitioning of order placement from order processing is possible
- Potential to re-use the order processing logic is lost, since it is not directly exposed as a separate service, but rather is directly invoked by the order placement service

TCP and Http Asynchronous Order Processing

As with WCF Web Services for the middle-tier business services calls from the Web application, the WCF Order Processing Service supports several distinct transport bindings represented by different OrderMode settings. The first two, ASync_Http and ASync_Tcp, are not based on messaging, but rather one-way asynchronous web service calls. The second two, ASync_Msmq and ASync_Msmq_Volatile, are based on messaging and a loosely coupled design for WCF services.

- Async_Http: Orders are processed by a WCF order processing service that is asynchronously invoked by the order placement service. The communication (binding) between services is Http, with text-xml order object encoding. The order processing service (TraderOrderHost.exe) can be co-located or remote, running on a single server or server cluster. Note that the order processing service must be online for users to be able to place orders. A rollback is issued

(effectively cancelling the order) if the order processing service is not online at the time of order placement in the browser (and the user is notified immediately).

- Async_Tcp: Orders are processed by a WCF order processing service that is asynchronously invoked by the order placement service. The communication (binding) between services is Tcp, with binary order object encoding. The order processing service (TraderOrderHost.exe) can be co-located or remote, running on a single server or server cluster. Note that the order processing service must be online for users to be able to place orders. A rollback is issued (effectively cancelling the order) if the order processing service is not online at the time of order placement in the browser (and the user is notified immediately).

In this mode, a service-to-service call is made: the order service asynchronously invokes (via a one-way service contract) the order processing service for each buy or sell order as requested by user input from the Web UI Layer. This mode is depicted below[8]:

[8] Neither of these modes corresponds to an existing WebSphere Trade 6.1 Order Processing configuration. For asynchronous order processing, Trade 6.1 supports a single Message-Driven Bean (MDB) asynchronous model that utilizes the IBM Service Integration Bus, SIB message queues, and JMS messaging to process orders asynchronously. The .NET StockTrader settings that do correspond (using .NET and Microsoft technologies) to Trade 6.1 asynchronous order processing are the ASync_Msmq and ASync_Msmq_Volatile settings. Please refer to the *Benchmarking .NET StockTrader vs. IBM WebSphere Trade 6.1* benchmark paper for details.

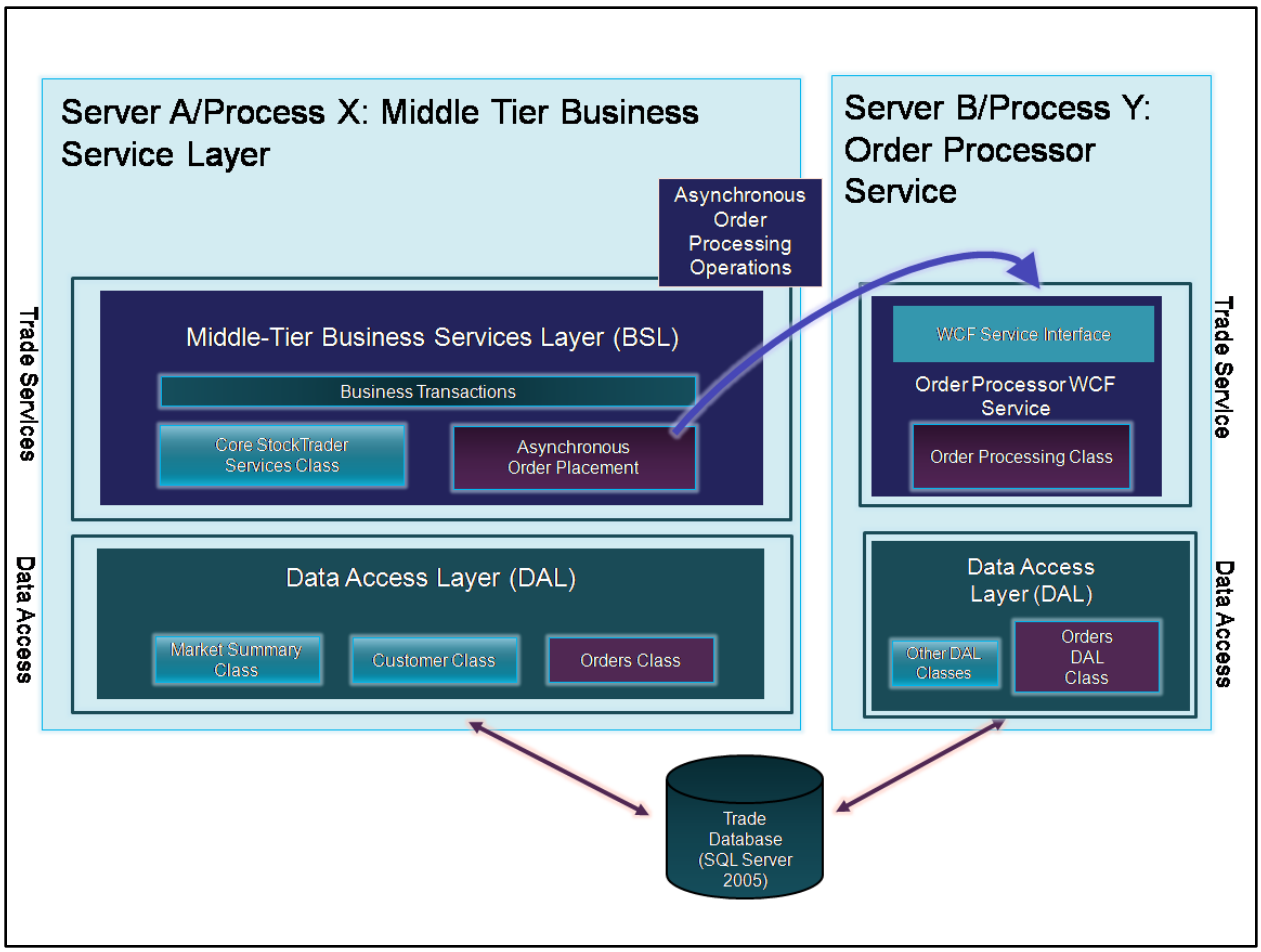

Figure 11: Asynchronous Order Placement via Tcp or Http and WCF Service

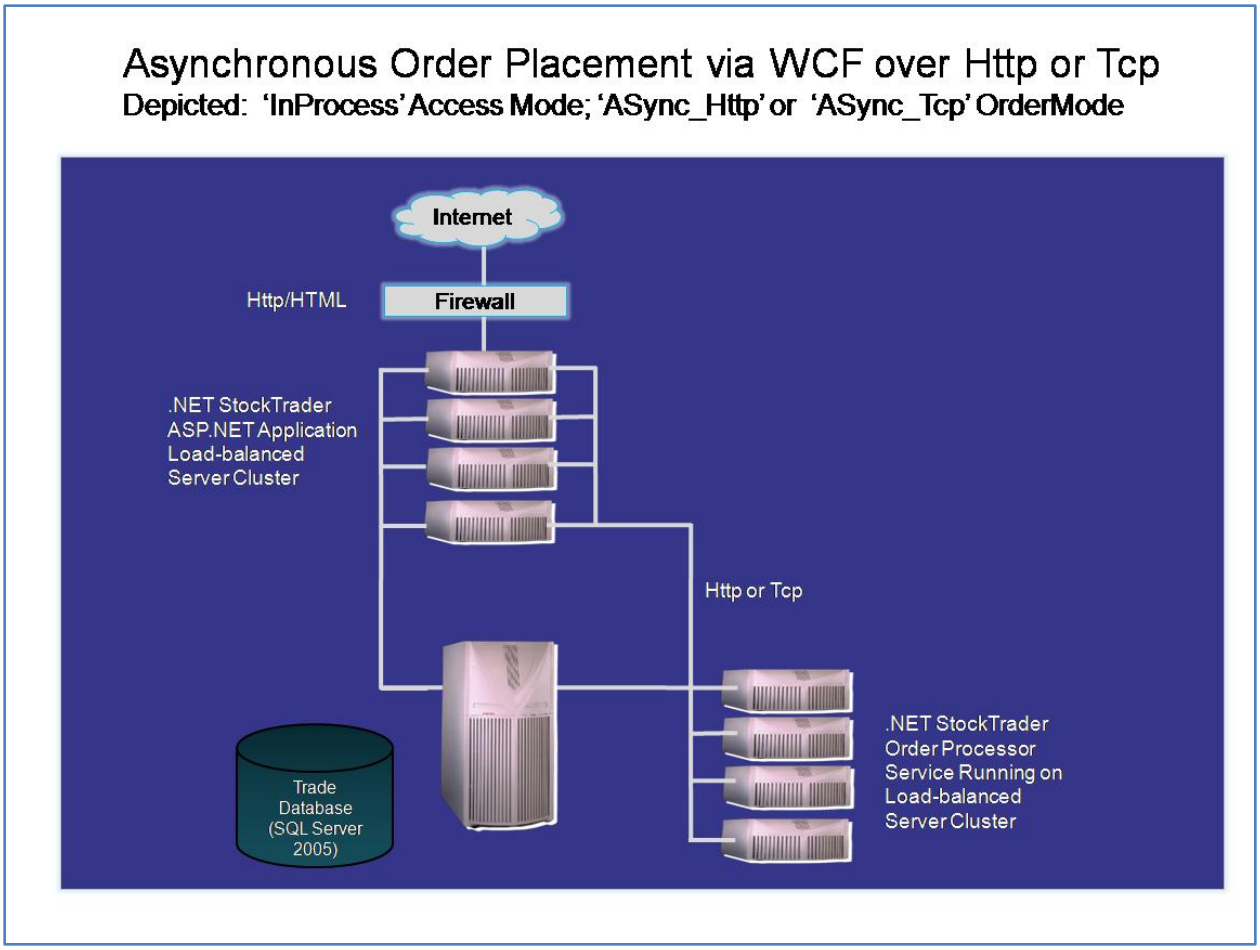

Figure 12: .NET StockTrader in Asynchronous order processing Mode with remote/asynchronous invocation of the order processing service via Http or Tcp binding

When running in Async order mode over TCP or HTTP, System.Transactions is used by the order processing WCF service to handle all work as an autonomous unit of work. In addition, the asynchronous order placement client wraps the creation of the order header (the order placed by the users via a UI-driven buy or sell operation) with the communication with the order processor service. If the order placement service cannot communicate to the order processor service, then the order is effectively cancelled with all work rolled back and no order header left in the database (the user is notified immediately). However, if the order is successfully communicated to the remote order processor service, but the order processor service cannot process the order (for example, a bad database connection or other exception); then the order header will be left in the database since these are two distinct One-Phase transaction (one by Business Services to create the header; the second to read the header and process the order in the Order Processing Service). Since the client is using a one-way service contract and is not waiting on a callback, the user will simply continue to see an Open order that has not been processed (and not marked as 'Completed'). The order header contains the information necessary to re-create/re-try the operation at a later time (although this logic is not part of the application). In addition, the order processor service will place the active order it cannot process into a durable poison message queue; again to be able to effectively save the order and re-try the processing

at some future time, perhaps after additional workflows and/or audits of the order take place in the system (again, this logic is not implemented; just the shipping of the order to the poison message queue is implemented).[9]

Design Considerations

Characteristics of such a design are:

- Potential to separate workloads with heavier order processing operations taking place on a dedicated order processing set of servers running the order processing service. This can provide additional scale.
- Faster response to the user since the order may take more than a few seconds (potentially minutes or hours to complete) in a real-world scenario. In this mode, users are not waiting for the order to be processed; they can continue their work in the StockTrader Web application; yet they will still receive real-time notification when the order is actually processed.
- Potential to build multiple user applications that use the common WCF .NET order placement service (service re-use).
- Vertical scaling via additional processors for the order processing service
- Horizontal scaling via load-balanced clusters of replicated servers for the order processing service
- Failover/high availability via balanced clusters of replicated servers
- Remote activation of the order service via Http, potentially over the Internet to a business partner
- Potential interoperability with Java/J2EE applications for asynchronous order processing
- Flexible, easier WCF programming model for building SOA applications
- Tightly coupled in that the order processor service must be online for users to place orders
- No assured message delivery

Note, however, while the system in such a configuration has fallbacks to protect and save orders that cannot be processed; this system does not represent assured message delivery. To achieve this, the .NET StockTrader can be run in Async_Msmq Order Mode; which represent a better design for a variety of reasons explained in the next section.

[9] This is functionally equivalent to the WebSphere Trade 6.1 application. If the order placement operation cannot communicate with the JMS WebSphere order processing MDB (for example, the JMS messaging service is unavailable), the order is cancelled with no order header left in the database. However, the Trade 6.1 application does not preserve the order in a poison message queue should the actual order processing fail.

WCF with MSMQ Asynchronous Order Processing

WCF has the ability to easily bind a service to an MSMQ transacted (durable) message queue. The programmer only programs to the service, and does not have to program any MSMQ-specific logic—either on the client or on the service itself. WCF handles the MSMQ interface automatically as part of the infrastructure. The WCF MSMQ binding works in conjunction with the WCF transaction model, using a service operation attribute to set the transactional attributes. In this case, a two-phase commit operation takes place when the service operation reads from the MSMQ queue and attempts to process the order. If the order processing logic fails, the order will remain in the queue and not be lost. WCF includes simple configuration settings for automatically re-trying the operation a number of times. The .NET StockTrader also implements a custom WCF behavior to handle poison messages automatically after the user-set retry limit is reached. In this case, the order is taken from the main message queue and placed into a separate poison message queue. The most important point is that orders, once in the system, are never lost, thus providing reliable, assured message delivery. This includes system failures, since the MSMQ queue is durable: all messages are persisted to disk. In a production setting, this might typically be used in conjunction with Microsoft Windows Server Clustering Services (included with Windows Server Enterprise and Data Center Editions) to provide fault tolerance with up to 8 backup servers ready to take over queue processing.

In this mode, the .NET System.Transactions infrastructure via the WCF transactional attributes again handles all the two-phase commit logic automatically, with no transactional or message queue programming required by the developer. This is depicted in the StockTrader code snippet below showing the Order Processing Service contract and operation contract.

```
[ServiceBehavior(ReleaseServiceInstanceOnTransactionComplete = false, TransactionIsolationLevel =
System.Transactions.IsolationLevel.ReadCommitted, ConcurrencyMode = ConcurrencyMode.Multiple,
InstanceContextMode = InstanceContextMode.PerCall)]
    [PoisonErrorBehavior]
    public class OrderProcessorService : IOrderProcessor
    {

.
.
.
.


    }


[OperationBehavior(TransactionScopeRequired = true, TransactionAutoComplete = true)]
   public void SubmitOrder(Trade.TraderOrderHost.QueuedOrder order)

   {
.
.
.
.


   }
```

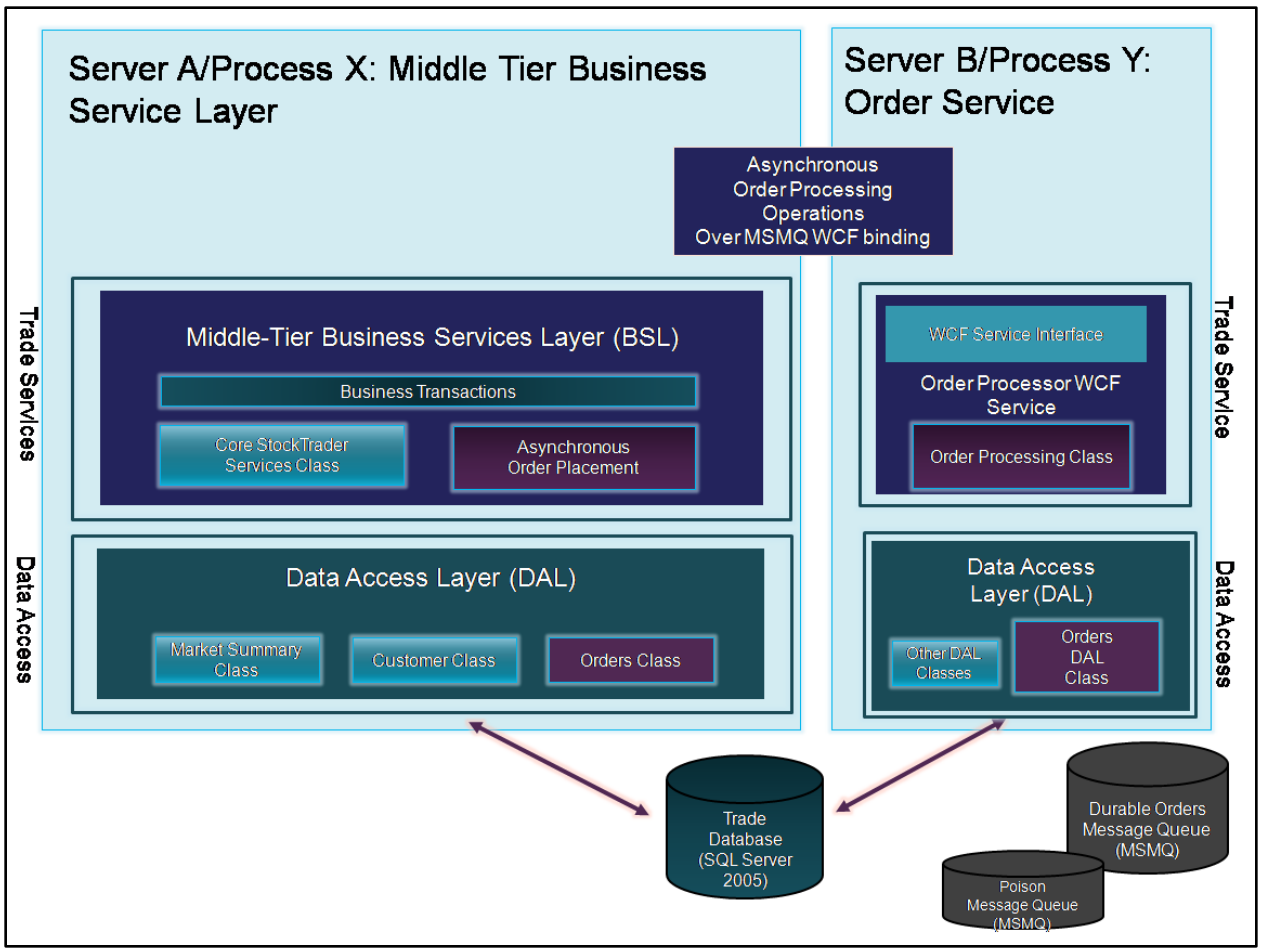

Figure 13: Asynchronous Order Placement via Msmq and Message-based Service

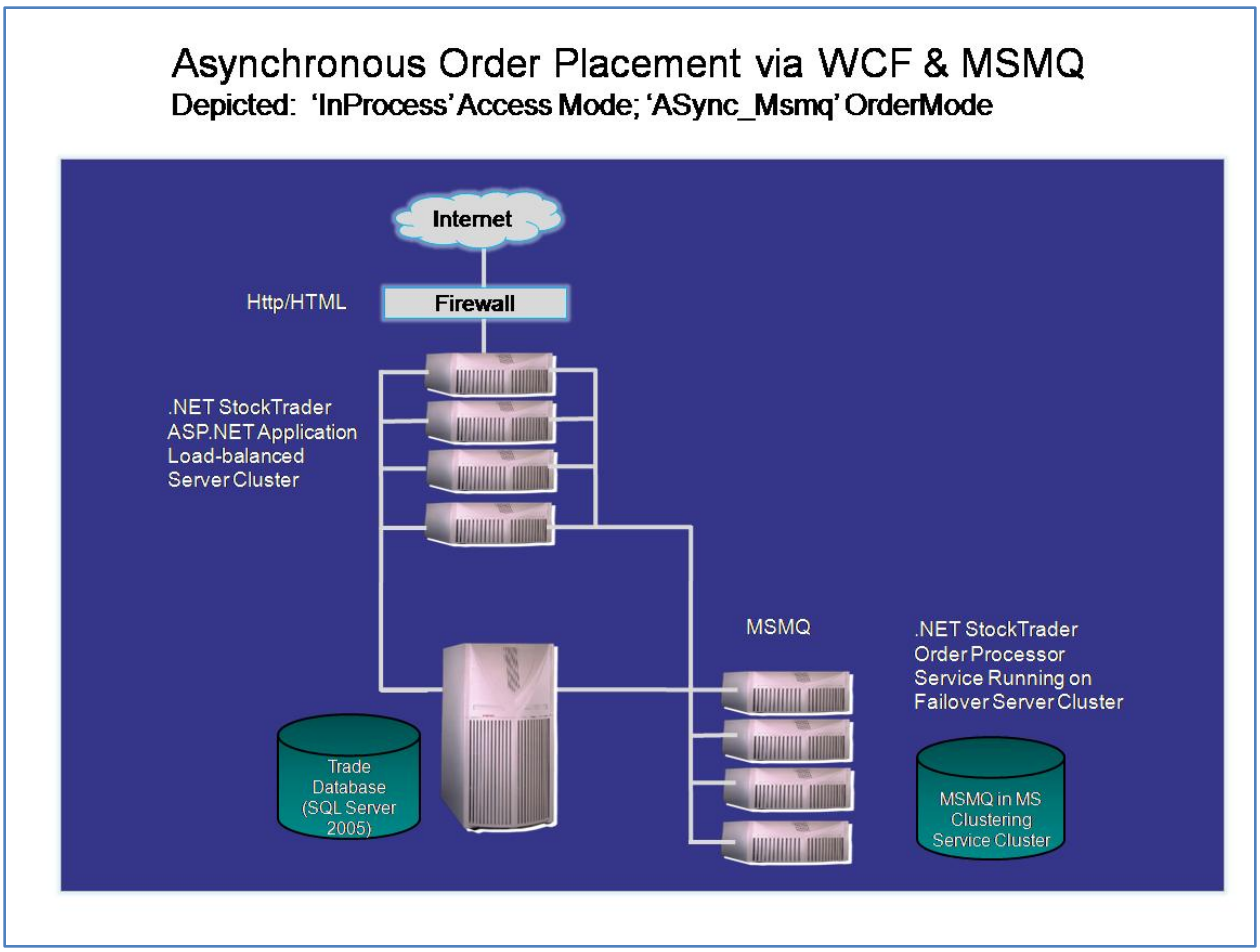

Figure 14: NET StockTrader in Asynchronous order processing Mode with remote/asynchronous invocation of order processing Service via Msmq messaging

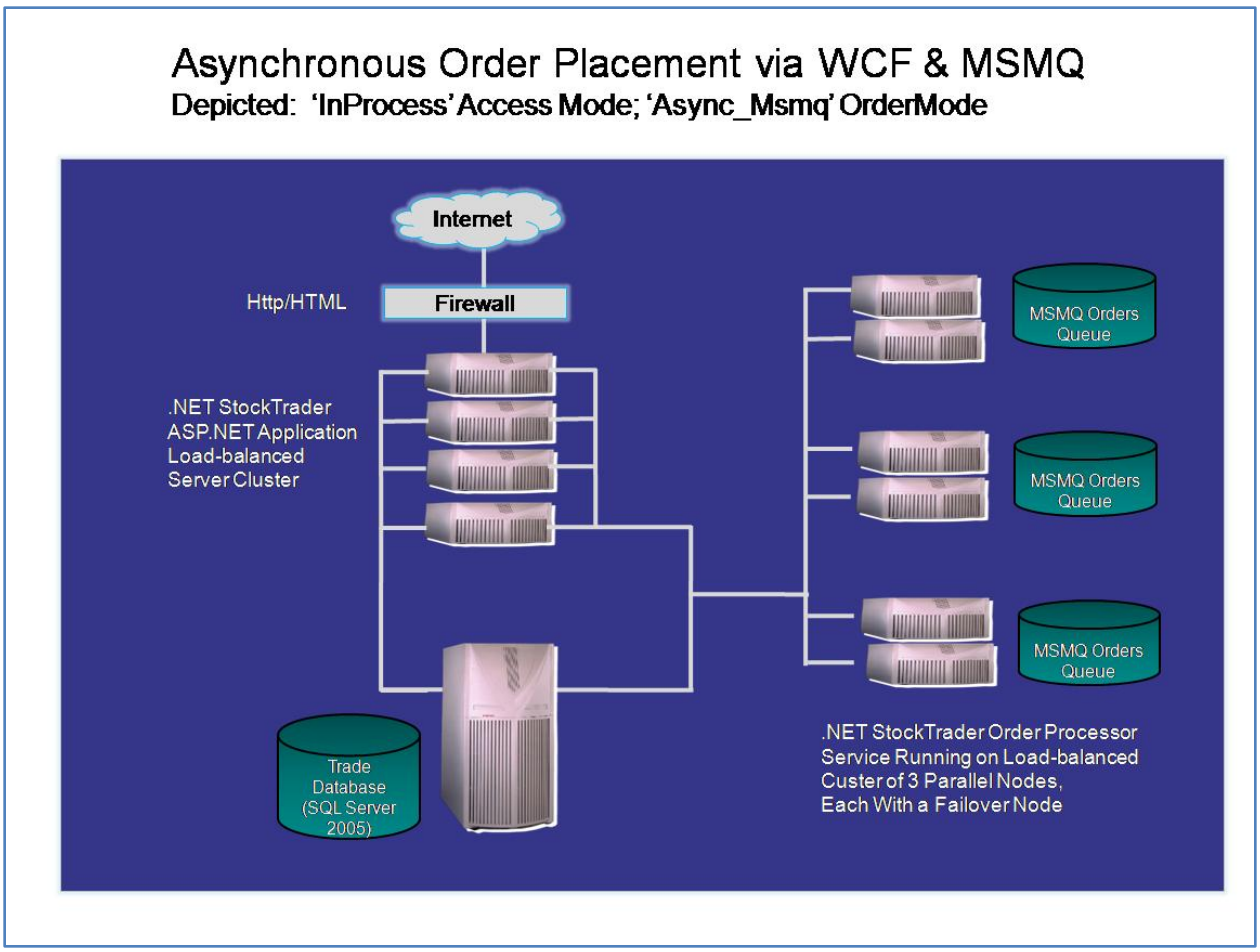

Figure 13: An alternative deployment topology: NET StockTrader in Asynchronous order processing Mode with remote/asynchronous invocation of order processing Service via Msmq messaging

Conclusion

The .NET StockTrader sample application explores several different alternative architectures for a service-oriented application. Implemented on WCF, it illustrates the flexibility of .NET 3.0 to support many different deployment topologies. The application illustrates many best-practice programming practices for .NET and WCF including the use of an n-tier, service-oriented design. As a benchmark sample and downloadable benchmark kit, the application also illustrates best-practice programming for building high-performance and scalable service-oriented applications using Microsoft .NET and WCF. Please refer to the benchmarking whitepaper on the StockTrader MSDN site for performance/throughput benchmark comparisons based on the application and the various modes it supports.