# WCF Math Service Library with Multi-Threaded Capabilities & Console Service Host and a Client “Tester” ChannelFactory Implementation

## About

This project presents a WCF Service Library (MathServiceLib) that consists of a math library that contains complex, long-running methods. This WCF Math Service and Client demonstrates multi-threaded access with various combinations of instance context modes and concurrency levels. The client accesses these methods using sequential and multi-threaded techniques to demonstrate the difference in performance.

The service is hosted via a console application (MathServiceHost). The client (MathClient) uses the ChannelFactory pattern as opposed to “Add Service Reference” with SVCUTIL. The client and service share a common assembly (SharedLib) that contains the key contract and data model information.

Furthermore, a Utilities project is used by the client console application to facilitate user data entry and the complicated details of building and managing the WCF ChannelFactory connection implementation. The ProxyGen class inside the Utilities project abstracts the details of implementing and managing a generic ChannelFactory connection to a generic service for a client. Note: The Utilities project library was included as base code for my lab project to facilitate speedy completion; we were not expected to code this Utilities project ourselves due to complexity and time constraints. The remaining projects in the solution (SharedLib, MathClient, MathServiceHost, and MathServiceLib), I completed individually per requirements for the lab project.

## Architecture

The MathService Visual Studio solution application consists of five project assemblies:

* SharedLib (Class Library) “Service and Data Contracts”
* MathServiceLib (WCF Service Library) “Implements the Service”
* MathServiceHost (Console Application) “Hosts the Service”
* MathClient (Console Application) “Client Tester to Service”
* Utilities (Class Library) “Helper classes”

## SharedLib

This library project contains the service contract required by this service. You can see the project code here.

### Service and Operations Contract

There is only one service interface defined: IMathService (interface for the client calling the service). You can see the IMathService code here has two operational contracts.

/// <summary>

/// IsPrime

/// Determines if a number is prime.

/// Works with very large integers

/// </summary>

/// <param name="number">(BigInteger) number to determine if prime</param>

/// <returns>(bool) true if a prime number, else false</returns>

[OperationContract]

bool IsPrime(BigInteger number);

/// <summary>

/// Sqrt

/// Determines the square root of a number

/// Works with large integers

/// </summary>

/// <param name="number">(BigInteger) number to determine square root</param>

/// <returns>(BigInteger) number of the square root</returns>

[OperationContract]

BigInteger Sqrt(BigInteger number);

## MathServiceLib

MathServiceLib contains the service code that implements the IMathService interface. It has a project reference to SharedLib. Click here to see the project code.

### MathService Class

This class implements the IMathService interface and has the ServiceBehavior attribute applied to it. The ServiceBehavior attribute has its InstanceContextMode property set to InstanceContextMode.PerCall to ensure each client call will receive a new service instance. In addition, the ConcurrencyMode property was *initially* set to ConcurrencyMode.Single. The testing procedure (later in this article) shows how changing these settings affects the solution performance results. Note: The *InstanceContextMode* set to PerCall ensures each client call will receive a new service instance while the *ConcurrencyMode* controls multi-threading access (Single is for Synchronous Access, Multiple = Multi-threading).

#### Sqrt(BigInteger value)

The Sqrt method calculates the approximate square root for an arbitrarily large integer value. By the way, that is what BigInteger provides – a data structure to hold numbers far larger that can be stored in traditional data types. Since the method uses the Math library, it is only going to be as precise as a double.

#### IsPrime(BigInteger value)

The IsPrime method uses a brute force (slow) algorithm to determine if a number is prime. There are some optimizations built in that eliminate Prime numbers.

## MathServiceHost

The MathServiceHost is a typical Console Application used to host the MathService service class. For the hosting configuration, I used the basicHttp protocol. The <system.serviceModel> portion of the config file was customized to the solution address, bindings, and contract. See the XML configuration here and project code here.

<system.serviceModel>

<services>

<service name="MathServiceLib.MathService">

<endpoint address="" binding="basicHttpBinding" contract="SharedLib.IMathService" />

<endpoint address="mex" binding="mexHttpBinding" contract="IMetadataExchange" />

<host>

<baseAddresses>

<add baseAddress="http://localhost:8733/MathService/" />

</baseAddresses>

</host>

</service>

</services>

<behaviors>

<serviceBehaviors>

<behavior>

<serviceThrottling maxConcurrentCalls="10" maxConcurrentInstances="10"/>

<serviceMetadata httpGetEnabled="True" httpsGetEnabled="True"/>

<serviceDebug includeExceptionDetailInFaults="False" />

</behavior>

</serviceBehaviors>

</behaviors>

</system.serviceModel>

## MathClient “Tester Client Console Application”

The math client project is a Console Application that consumes the service through a ChannelFactory pattern with the IMathService interface. This client program demonstrates multi-threaded access with various combinations of instance context modes and concurrency levels. The client accesses these methods using sequential and multi-threaded techniques to demonstrate the difference in performance. The goal of this client is to test a batch of calls to the service using a traditional sequential loop and a parallelized loop. Results of the performance tests are printed to the console window. You can see the project code here.

### Class Fields

The Program class will have several fields:

static IMathService m\_Proxy = null;

static int m\_Tests = 250;

static BigInteger m\_StartNum = BigInteger.Parse("1234567890123456");

*m\_Proxy* is the proxy object that connects to the service.

*m\_Tests* holds the number of iterations to perform per test.

*m\_StartNum* holds the initial number to use in each loop when calculating square roots.

### Class Methods

The Program class contains three static methods: Main, TestSync, and TestAsync.

#### Static TimeSpan TestSync()

The TestSync method calls the service method *IsPrime* in a synchronous manner. It iterates for a set number of times, adding one to the original number and determining if that number if prime or not. The method accepts no parameters and returns a TimeSpan object which will contain the total time spent in the method.

#### Static TimeSpan TestAsync()

The TestAsync method does the same thing as TestSync except it will use a Parallel.For loop to call the service method from the thread pool.

#### Static Void Main()

Firstly, the client proxy channel to the service, m\_Proxy, is initialized using the *Utilities* project (helper classes) for creating and managing a ChannelFactory pattern connection.

The console application then asks the user to enter new values for m\_Tests (number of iterations to perform per test) and m\_StartNum (the initial number to use in each loop when calculating square roots). This helps the program stay flexible for testing various ranges and values.

To ensure that both synchronous and asynchronous tests are treated “equally”, a quick call to the service is made to “spin it up”. That way when the tests are executed there is no initial delay with one of them starting the service for the first time.

Next, calls to both the TestSync and TestAsync methods are made. Their results are the performance of the methods executing either in synchronous or asynchronous mode. The results stored in two different TimeSpan objects.

Finally, the console program outputs to the user the TotalSeconds value from each of the TimeSpan objects with an appropriate header for each. The user can then see the apparent difference in performance between the synchronous and asynchronous calls, if any.

### Configuration

The client app.config file was updated with the correct serviceModel configuration. See the XML configuration file here.

<system.serviceModel>

<client>

<endpoint address="http://localhost:8733/MathService/"

binding="basicHttpBinding"

contract="SharedLib.IMathService"

name="basicHttpBinding\_IMathService"/>

</client>

</system.serviceModel>

## Utilities

Furthermore a “Utilities” project is used by the client console application to facilitate user data entry and the complicated details of building and managing the WCF ChannelFactory connection implementation. The “ProxyGen” class inside the Utilities project abstracts the details of implementing and managing a generic ChannelFactory connection to a generic service for a client. Note: The Utilities library was included as base code for my lab project to facilitate my speedy completion; we were not expected to code this ourselves due to complexity and time constraints. The remaining projects in the solution (SharedLib, MathClient, MathServiceHost, and MathServiceLib), I completed individually per requirements for the lab project. Click here to see the Utilities code.

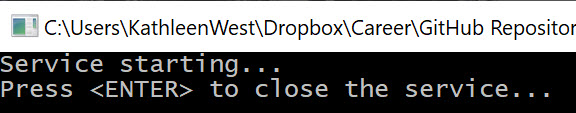
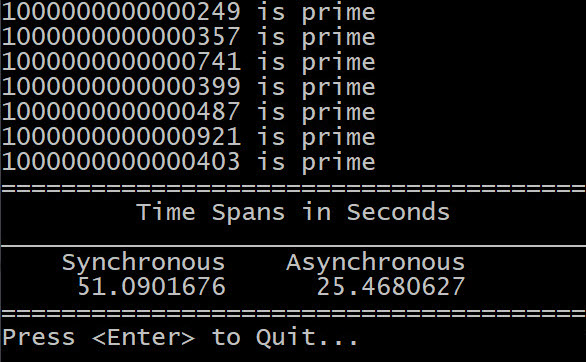
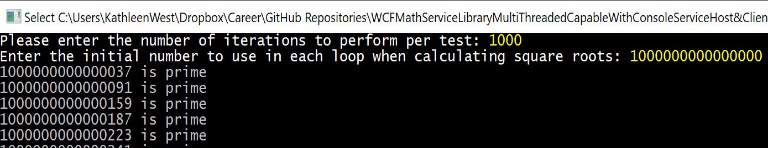
## Testing the Service Performance

The table below presents test results for running the service with different parameters. The InstanceContextMode on the service was set to *InstanceContextMode.PerCall* to ensure *each client call will receive a new service instance*. Next, the test was run either with the service ConcurrencyMode marked either as Single (not capable of multi-threaded) or Multiple (multi-threaded capable). Lastly, two additional *service throttling* behavior configurations were investigated: Max Concurrent Calls and Max Concurrent instances that were set via the App.config file for the MathServiceLib (project that implements the service). These two options limited or “throttle” the traffic flow on the service.

You can learn more about using ServiceThrottlingBehavior to control WCF Service performance here. [https://docs.microsoft.com/en-us/dotnet/framework/wcf/feature-details/using-servicethrottlingbehavior-to-control-wcf-service-performance]

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Test # | Concurrency Mode | Max Concurrent Calls | Max Concurrent Instances | Synchronous Time (s) | Asynchronous Time (s) |
| 1 | Single | 1 | 1 | 5.4841596 | 3.1109451 |
| 2 | Single | 32 | 1 | 4.5342732 | 4.4764453 |
| 3 | Single | 1 | 32 | 4.5110034 | 3.4880489 |
| 4 | Single | 32 | 32 | 4.6642406 | 3.4156655 |
| 5 | Multiple | 1 | 1 | 4.7430622 | 3.8245755 |
| 6 | Multiple | 32 | 1 | 4.4457601 | 3.4779633 |
| 7 | Multiple | 1 | 32 | 4.3398398 | 3.440013 |
| 8 | Multiple | 32 | 32 | 4.529524 | 3.4022743 |
| 9 | Multiple | 50 | 50 | 4.713197 | 3.3144912 |
| 10 | Multiple | 100 | 50 | 4.7247299 | 3.3956434 |
| 11 | Multiple | 50 | 100 | 4.7948504 | 3.4389735 |
| 12 | Multiple | 100 | 100 | 4.7498103 | 3.4859596 |

## Demo



## Code

The entire project code repository is available on GitHub here.