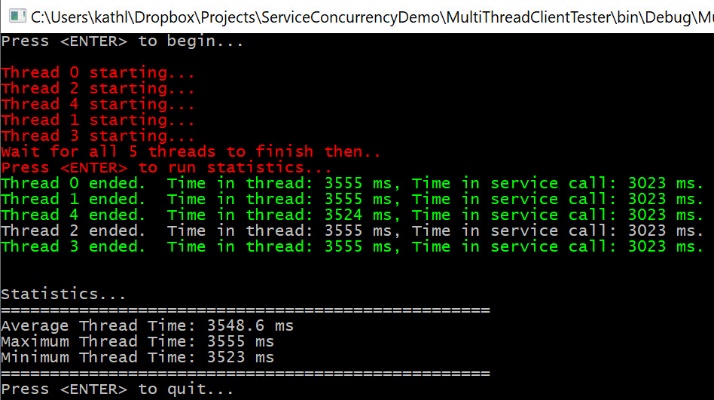
# Concurrency and Instancing in WCF Services – A Demo of Service Concurrency and Instancing Behaviors with Multi-threading Clients

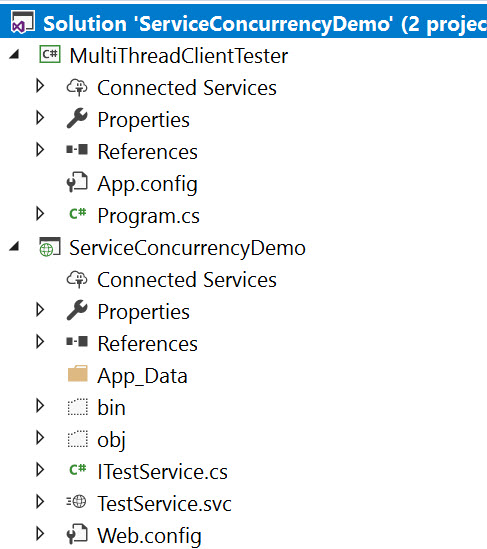
## 

## About

This project presents a simple Demo Service and WCF “Tester” Client Application demonstration that implements concurrency and instancing behaviors on a service with multiple client thread calls to a method on the service. The Demo Service is a standard template WCF service application hosted by the development IIS. The service features one simple method… a test method that simulates a long running process (it sleeps for 3 seconds). The client “tester” is a simple console application that creates multiple threads that access the service and report back on the results. The objective of this project was not to demo setup and hosting of a service, nor the client interface, but retrieve and display results of service behaviors with respect to multi-threaded access. Discussion regarding the hosting and setup of the simple IIS hosted service application will be skipped in this project article.



## Architecture



The demo project consists of these simple component topics:

* WCF Service (Hosted by IIS) Application Project “ServiceConcurrencyDemo”
  + Program (Starts, Manages, Stops the Service)
  + Web.config (Configuration for Service Hosted on IIS - Generic)
* Client “Tester to Service” Windows Console Application “MultiThreadClientTester”
  + *Service Reference Proxy (WSDL) to WCF Service*
  + Program – Creates multiple threads that call the service, records time spent in threads, and reports statistics on the final results in a console window.

## Service Application

A service template application project (*ServiceConcurrencyDemo*) was added to my Visual Studio solution. The code is available on GitHub [here].

### ITestService (Interface for Service)

This interface is a simple test service that simulates a long running process. The ServiceContract for the simple demo service contains only one operation contract. The method simulates a long running process on a service and returns the TotalMilliseconds (int) spent in the service method. The code is available on GitHub [here].

[insert code here]

### TestService.svc.cs (Code that Implements the Service Interface)

The service implementation code has service-level attributes configured to control the concurrency and the instance behaviors of the service. This code implements the contracts for the ITestService interface. The code is available on GitHub [here].

There are four modes that were available for configuration on the service:

* Concurrency: Multiple | Instance: PerCall (Not Relevant)
* Concurrency: Multiple | Instance: Single\*
* Concurrency: Single | Instance: PerCall (Not Relevant)
* Concurrency: Single | Instance: Single

*\* This is the current configuration on the code in the repo.*

The definitions, results, and discussion of the service behaviors is discussed in more details in the results section. The use of concurrency is related to the instancing mode. In PerCall instancing, concurrency is not relevant, because each client call to the service is processed by a new service instance.

[insert code here of the attributes]

#### Service Interface Implementation Methods

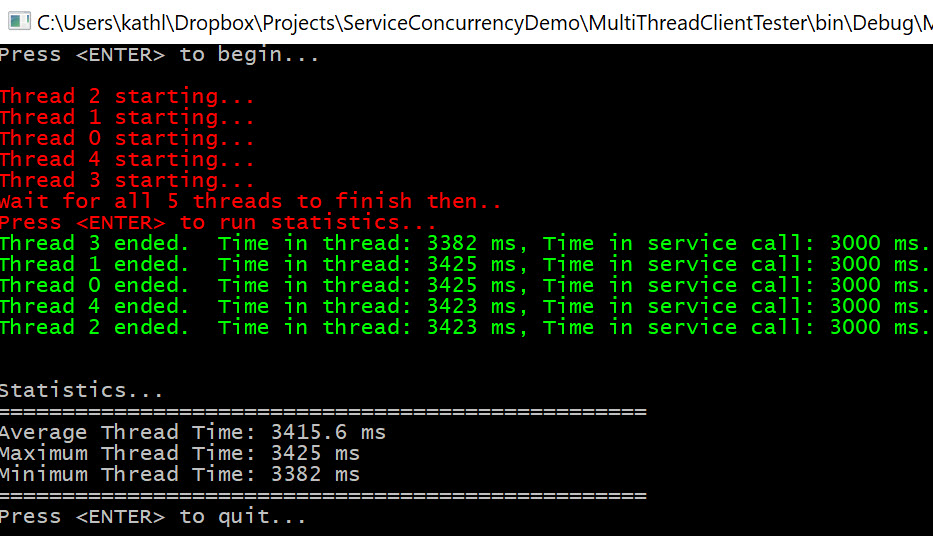
The service implementations are described below.

##### TestMethod

This is the only method available on the service. When a client calls this method, it simulates a long running process by sleeping for three seconds and then returning the time spent in the method as an integer representing milliseconds.

[insert code here]

## Client “Tester to Service” Windows Console Application



The client “tester to service” is a simple windows console application project (*MultiThreadClientTester*) in the same solution that connects to the “Demo Service” by use of a Service Reference proxy generated by the Add Service Reference Wizard. The client program will use this proxy to test the OperationContract or methods available in the ServiceContract and return the results to the user on the console window.

The project code is available on GitHub [here].

### Features

The client application was meant to only demonstrate consumption of the Demo service, so this client has limited scope of features. Some of the client application features are:

#### Simple Design

The user interface is simplified to do basic demo, testing, and reporting of the multi-thread calls to the service with its configured behaviors.

#### Multi-threading

The program creates and runs threads while each one calls and waits on a call to the service in parallel with other threads. Each thread asynchronously updates its run time on the console window after it completes the service call. The program flow is actually asynchronous but asks the user to wait for the work to be complete before running the stats.

#### Stats

After the thread work is complete, the user presses enter to displays simple stats on the threads including the average, max, and min run-times of the threads.

### Program Code

The code behind file is for the client tester and manages the application in one file. The code behind the form is available on GitHub [here].

#### Fields

Private fields allow the client to work with the following:

* Proxy for the Service Reference (Generated by the SVCUTIL Wizard)
* List of integers representing results of thread run-times

[insert code here]

#### Methods

##### Main

This main method is the entry point of the client tester application. It first creates a proxy object (instance of the reference created by SVCUTIL), then creates and runs 5 threads in parallel. It then asks the user to wait until thread completion so they can see the individual results on the console window and then display the stats.

[insert code here]

##### ServiceThread

This is the main thread that implements the *“async await”* pattern of asynchronous processing of the thread. It also accesses the service using the async version of the generated proxy methods (proxy.TestMethodAsync()). It calls the service and awaits the completion, then adds its total thread time information to the list of thread times.

[insert code here]

##### Print Statistics

This method runs and prints basic statistics (average, maximum, minimum) on the List of thread times.

[insert code here]

## Results

Recall there were four modes available for the service configuration:

* Concurrency: Multiple | Instance: PerCall *(Not Relevant)*
* Concurrency: Multiple | Instance: Single\*
* Concurrency: Single | Instance: PerCall *(Not Relevant)*
* Concurrency: Single | Instance: Single

*\* This is the current configuration on the code in the repo.*

### Definitions

#### Concurrency Multiple

#### Each service instance processes multiple client method calls concurrently. The service implementation must be thread-safe to use this concurrency mode. This is ideal for multiple threads accessing the service method at the same time, however, resources must be thread-safe. The use of concurrency is related to the instancing mode. In PerCall instancing, concurrency is not relevant, because each message is processed by a new service instance.

#### Concurrency Single

#### Each service instance processes one client call to a method at a single time. This is the default concurrency mode (if not manually configured with attributes). If there is a multi-threaded client having multiple threads call this service, then the client call thread requests will be processed in series. Each service caller/thread must wait until the service has completed the current caller it is processing. The use of concurrency is related to the instancing mode. In PerCall instancing, concurrency is not relevant, because each message is processed by a new service instance.

#### Instance Single

In this mode, only one service InstanceContext object is used for all incoming client calls and is not recycled after the calls. If a service object does not exist, a new one is created when the service is called for the first time.

#### Instance PerCall

In this mode, a new service InstanceContext object is created prior to and recycled subsequent to each client call.

### Summary

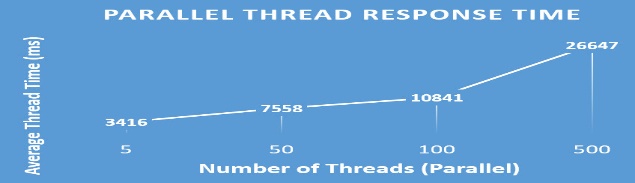
Results of average thread performance for the four modes are summarized below

* Concurrency: Multiple | Instance: Single\* | Fastest | 3417 (ms)
* Concurrency: Multiple | Instance: PerCall *(Not Relevant)* | Medium | 3549 (ms)
* Concurrency: Single | Instance: PerCall *(Not Relevant)* | Medium | 3651 (ms)
* Concurrency: Single | Instance: Single | Slowest | 9472 (ms)

*\* This is the current configuration on the code in the repo.*

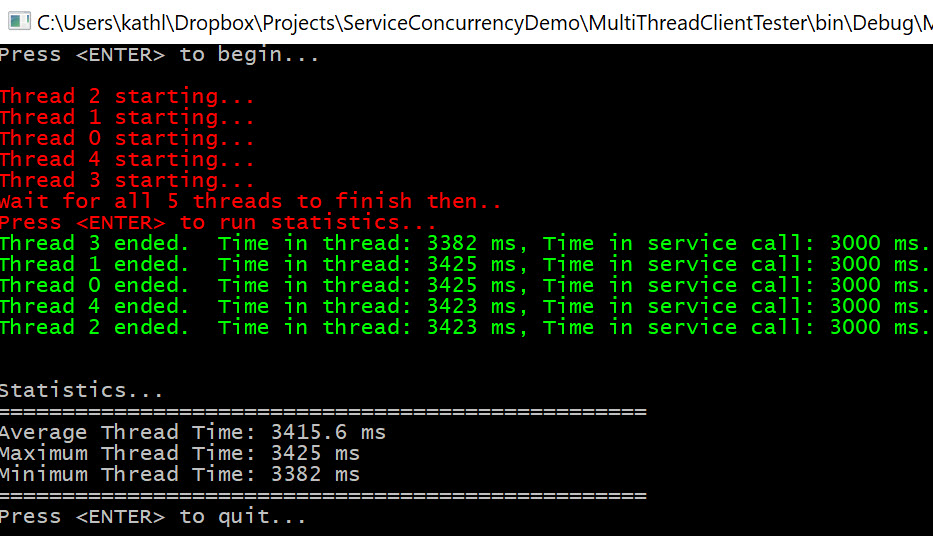
### Parallel Thread Response Results

The user can take this demo project one step further and increase the number of client threads to run in parallel while the service is configured to run concurrency multiple on a single instance. This will truly test the service capability and performance. The diagram below shows the results of average thread performance times as the number of threads increased from 5 to 50, 100, and 500 respectfully.

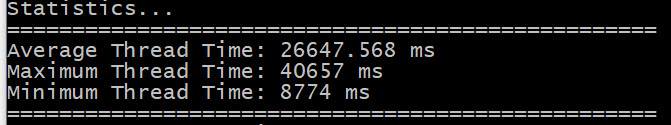


## Demo

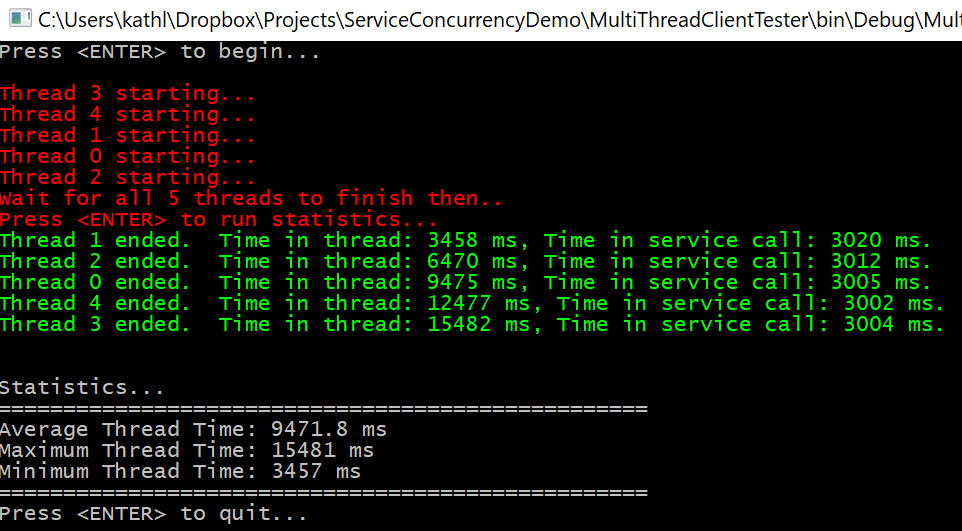
The client console application calling the service configured in concurrency multiple mode with a single instance serving 5 threads.



Serving 500 simulataneous threads the results are:



The client console application calling the service configured in concurrency single (no multi-threading allowed) mode with a single instance serving 5 threads.



## Code

The entire project code repository is available on GitHub [here].