# The Money Pot Problem – A Demo of Concurrency in Singleton and PerCall Service Behaviors in a WCF Service Application & Client "Tester" Windows Form Application

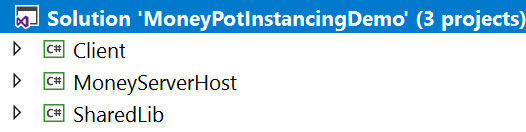
## 

## Problem Formulation

Pretend there is a fictional money pot existing unguarded on a computer server. The leprechauns have left their pot of gold unguarded and anyone can take $ money from the pot. But there is a catch: the clever elves have limited the amount of cash that can be taken per transaction request and guarded the resources to prevent multiple money grabs at the same time (resource thread locking). In a race to get all the money available from the money pot, several computer fairies are hacking into the money pot and competing against each other to get the most money. Each fairy starts their computer “client” to connect to the leprechauns’ money pot server to drain as much as they can. To their disappointment, the computer fairies find that the more client connections (competition from other hackers) and requests they make to the money pot, the less $ money they receive compared to what they would receive if only one fairy hacker was hacking the money pot.

## About

This project presents a simple, but fun “Money Pot” Service and Client Application demonstration. The “Money Pot” is a self-hosted (service host) WCF application with a GUI user interface to quickly demo and test the service with a client (both simple Windows Form Applications).



The service interface is defined not in the service application but in a *Shared Library*. This library defines the interface contracts for the $ money withdrawals and is referenced by both the client and service host projects.

A client “tester” windows form application tests the service and provides output to the user in a simple GUI.

In addition, a short discussion of concurrency to protect resources against multiple threads is shown along with charts, pictures, and test data to show that with multiple threads (clients) wanting the “money pot” resource, can diminish what resources an individual client thread can receive if they compete against one another.

## Introduction & Terminology

### Service Behaviors

In software engineering, the singleton pattern is a software design pattern that restricts the instantiation of a class to one "single" instance. In the “money pot” service behavior design there were two options studied:

**Single:** One Singleton instance is created for the server and all client calls.

**PerCall:** Each client call spins up a new service instance.

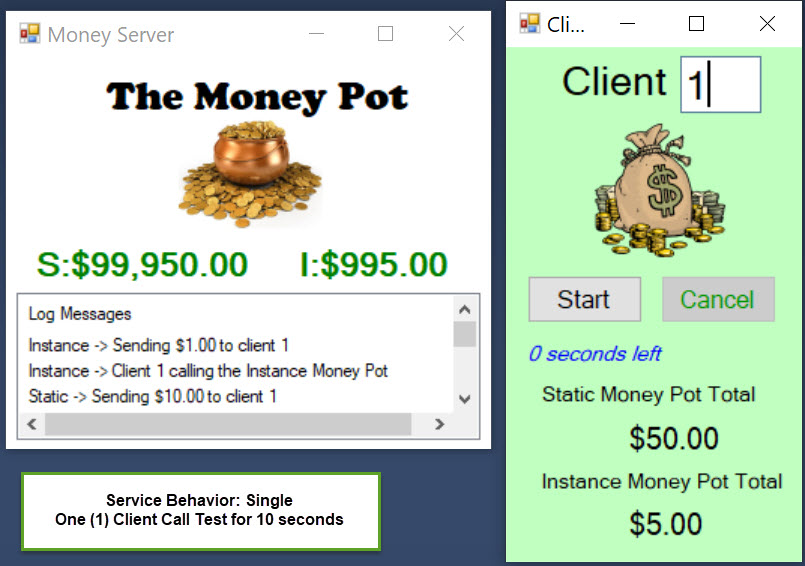
You can learn more about WCF Service Behaviors here [<https://docs.microsoft.com/en-us/dotnet/api/system.servicemodel.instancecontextmode?view=netframework-4.8>]

### Service Data

**Static:** When the service spins up on the host it creates an instance on the server. This is true for either the Single or PerCall behavior. The static data is accessible on the class level instance. Any method calls or changes to static class data will be seen by all clients.

**Instance:** When the service spins up on the host it creates an instance on the server. This is true for either the Single or PerCall behavior. However, in the Single service behavior the static class and instance are shared by all clients. In the PerCall service behavior, the clients have their own private instance data that is inaccessible by other clients. The PerCall clients may still have access to modify the same static class methods and data properties “shared” by all clients just like the Single service behavior.

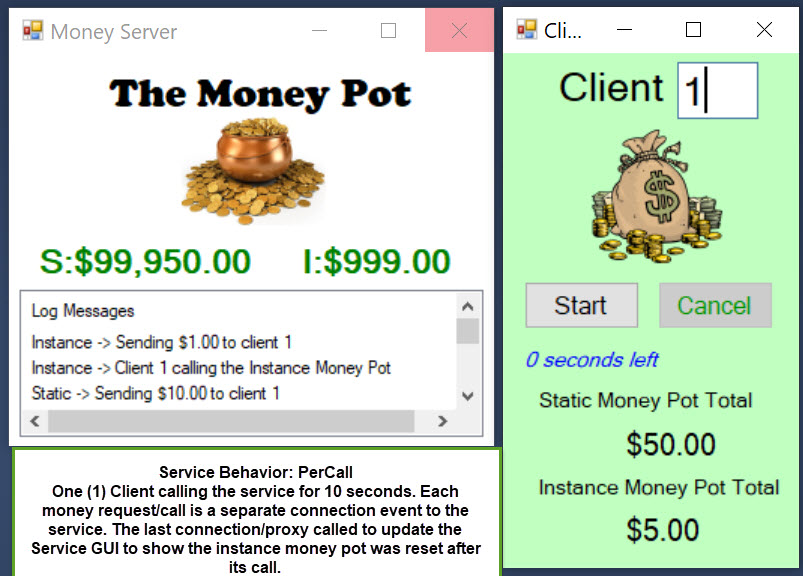
### The Money Pots



There are two money pots defined in the service: static and instance.

**Static Money Pot**: The static money pot is one pot of $ money that will decrease over time as clients call the service and ask for money from the static money pot. This decrease in resource money will occur in either the Single or PerCall service behaviors.

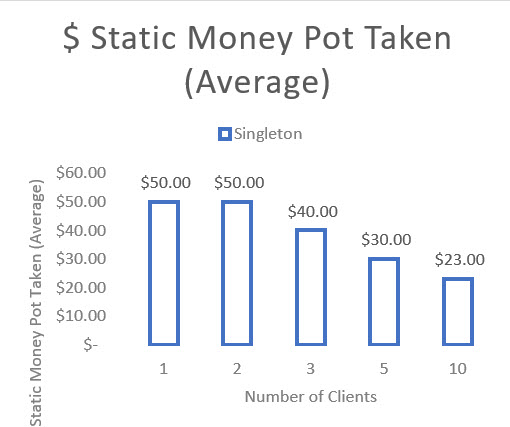
**Instance Money Pot**: The instance money pot is also one pot of $ money resource that will behave differently depending on the service behavior configuration. If the service behavior is Single, then the instance pot of money will decrease over time with the multiple client calls and it will behave as though the clients are pulling $ money resources from a global or static money pot. However, if the service behavior is configured to *PerCall*, then the instance money pot resource is private and accessible only to that client who called it. If there is $1000 in the instance money pot, the client who called the instance money would have access to the entire pot. Furthermore, each time they called the service (new connection), they would see the full money resource regenerated and available (another $1000 available). If a client called the service (PerCall) and got $900 out of the instance money pot, disconnected, then called the service again, it would have $1000 available in the instance money pot, not $100.



## Results Summary

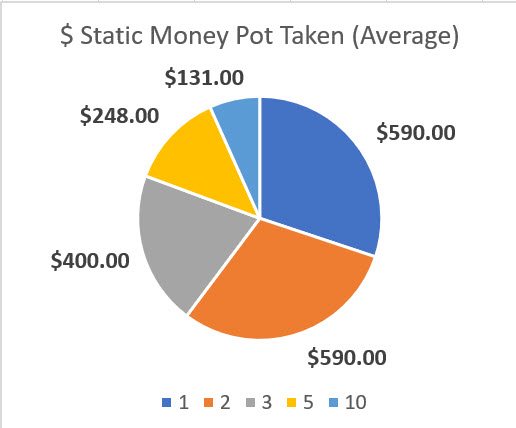
I ran my service host in Single mode testing the effect of multiple clients on their ability to withdraw money from the static money pot. My code and tests simulated a thread lock of 1 one second on the static money pot as each client accessed it, others would wait while the ThreadPool would be in charge of client thread access under the safety of the thread lock.

Below is a summary of the average amount of money received per client given the number of clients trying to simultaneously access the money pot *for a test run-time duration of ten seconds*.



As you can see, the more clients that accessed the service, decreased the average money pot $ resource received by the client due to the thread locks. Note: I manually “fired” the start button as fast as I could on the individual test clients and there was some slight delay to be understood in that click/starting event that puts the clients I started first as getting more resources than those clients started later after my click delay. This is why I compare the averages.

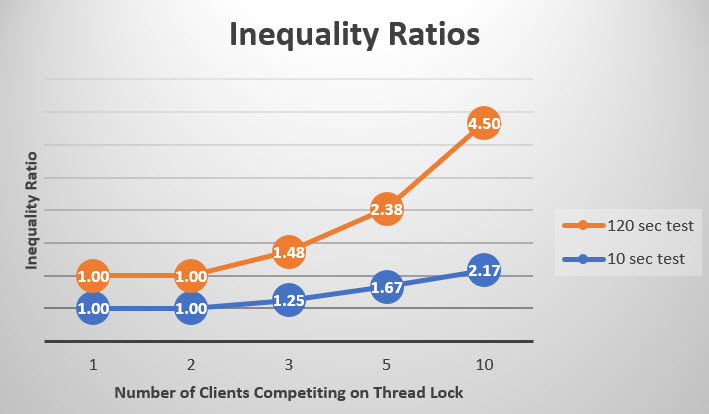
Next, the test duration was increased to *120 seconds* and I computed the average $ money received by each client for the given test case.



Once again, the average resource $ money received by a client *decreased* as competition increased.

### Inequality Ratios

The 10 second client test had an inequality ratio of 50/23 or *2.17* with 10 clients while the 120 second client test had an inequality ratio of 590/131 or *4.50* with the same number of clients running. This meant that the longer the time duration that the clients tried to access the static money pot resource (protected by a 1 second thread lock) the more inequal their money pot share (average) became.



## Architecture

The demo project consists of these component topics:

* WCF Service (Self-Hosted) Application Project “MoneyServerHost”
  + MoneyPotService (Code that Implements the Service Interface)
  + App.config (Configuration for Service Host)
  + *Reference to the Shared Class Library*
  + Main GUI (Windows Form Application)
  + Form Code – Processes GUI User Interface
* Shared Class Library Project “SharedLib”
  + IMoneyPotService (Interface for Service)
* Client “Tester to Service” Windows Form Application Project “Client”
  + *Reference to the Shared Class Library*
  + Main Form GUI User Interface
  + Form Code – Processes GUI User Interface

## Shared Class Library

A Class Library project (SharedLib) was added to my Visual Studio solution. The code is available on GitHub here [].

### IMoneyPotService (Interface for Service)

The ServiceContract for the Money Pot Service allows for two possible options: 1.) get money from the static money pot or 2.) get money from the instance money pot. The OperationContract will return an amount of money (decimal) to the client requester who is required to send them their client identification (integer). The service sends a fixed amount of $ money to the client. The code is available on GitHub here [].

[insert code here]

## WCF Service Application

A WCF Service (Self-Hosted) Application project was added to my Visual Studio solution. The code is available on GitHub here [].

### MoneyPotService (Code that Implements the Service Interface)

The service implementation code controls thread access (locks) to the money pot resources and returns a fixed amount of money with each request for access to the static money pot or the instance money pot. The service references a shared class library called “SharedLib” that contains the interface definition for the service. The code is available on GitHub here [].

#### Fields

The fields section contains object locks for concurrency, constant limits on the money pot withdrawals, and the money pot static and instance values. The static windows form GUI object is also defined here but instantiated in the main entry.

[insert code here]

#### Main Entry

The main program entry for the application is in the service implementation. It instantiates a new form window (GUI) that is accessible from the service and runs it.

[insert code here]

#### Service Implementation Methods

Both the service implementations for the static and instance money pot contracts follow a similar pattern: 1.) Update the service message log 2.) Lock access to the Money Pot resource with a thread lock object 2.) Simulate a one second delay (sleep) 3.) If there is money left in the pot, withdraw the limit amount from the resource, update the GUI, update the message log, and return the result to the caller or else, return the remainder of the money pot (which could be zero) and do the same process of updates.

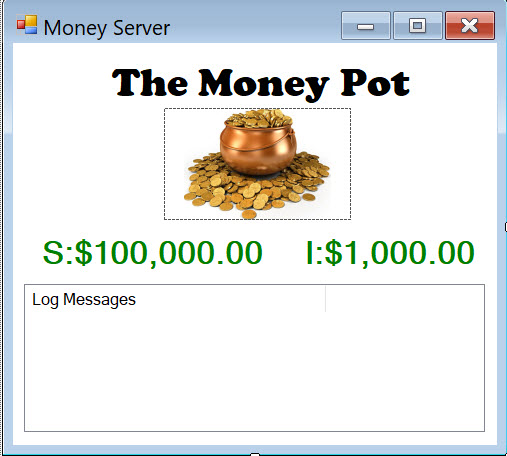
[insert code here]

### Application Configuration App.Config

The application configuration simple sets up the service model configuration including the endpoint, interface contract name, basic http protocol, and the base address for this demo. The code is available on GitHub here [].

[insert code here]

### GUI (Windows Form Application)



The Money Pot Server GUI is a simple Windows Form application with the intent of displaying the real-time dashboard of its static and instance money pot values during the demo tests. The code is available on GitHub here [].

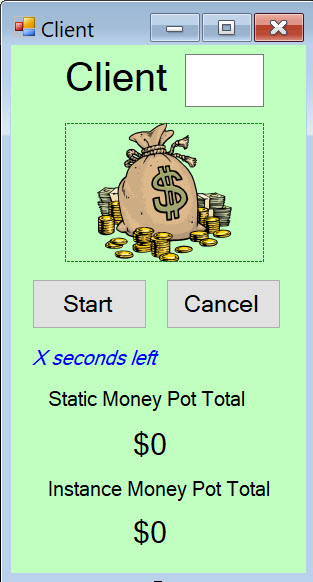
### GUI (Windows Form) Code

Code works behind the scenes to support the Windows Form GUI from launch to stop. When the form loads, it instances and opens a new Service Host (type MoneyPotService). When the form is closed, this service host is closed and the resource properly disposed.

The remainder of the form code handles the update of the money pot labels and log message data on the form list box. It receives the values to update from the business layer “Money Pot Service”. The code is available on GitHub here [].

[insert code here]

## Client “Tester to Service” Windows Form Application



The client “tester to service” is a simple windows form application project in the same solution that connects to the “Money Pot Service” by use of a ChannelFactory proxy to a specified service name that is defined in the client application configuration. 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 form window.

The client application shares and references the “Shared Library” – a class library project. It has access to the Money Pot service interface to know exactly how it can consume the service with the help of a ChannelFactory proxy (discussed in the next section). The project code is available on GitHub here [].

### Form Code Features

The code behind the form is available on GitHub here [].

#### Client ID Textbox

The user must enter a valid integer into the text box before the client program will connect to the service. It does not check the id against other clients (this is just a simple demo program).

#### Start Button

When the start button is clicked the program launches to the button clicked event that starts a sequence of process including a new thread separate from the UI.

1. The process is to first to try and parse the client id from the textbox entry before going further. If it detects an invalid entry, it notifies the user with pop-up window and returns to the main GUI. If the data entry is valid (an integer) and parsed, then the program sets the client id field and continues.
2. GUI is Updated: The start button is disabled and the cancel button is enabled. The run time countdown counter is set to show the user how many second the test will run (default 10 seconds) from the constant field value.
3. Time Elapsed and Timer fields are reset/initialized for the test run time.
4. A cancellation token source is initialized for the “Cancel” button or cancel event scenarios in the program.
5. Task Parallel Library (TPL) is setup to create a new thread for processing the call to the service with instructions and settings for what to do when the task is complete or cancelled.
6. Action is now passed to the CallService() method on a new Task (parallel)
   1. A ChannelFactory object of type IMoneyPotService is created and opened.
   2. A service proxy object is created from the ChannelFactory object
   3. The Timer is Started
      1. This time object calls the OnTimedEvent() method every 1 second. The event increments a time elapsed property. It then computes the run time remaining. If there is additional time remaining, it updates the UI Thread GUI (see notes on the cross-thread synchronization). If the time has run out: it stops the timer, disables it, and also updates the UI Thread GUI.
   4. The program runs while the timer is enabled
      1. At every iteration we will check to see if cancellation was requested and throw a new *OperationCanceledException* if so requested.
      2. The program tries to connect to both the static and instance money pots on the service and get money out. If successful, it updates how much money it has received in its total fields.
      3. After the money grab is successful, the client updates its UI GUI thread (see notes on the cross-thread synchronization)
   5. Exceptions and the cancellation requests are handled.
7. The task is now complete or cancelled.

[insert code here]

#### Cancel Button

If the user manually cancels the task or the program has a scenario to cancel, it will call the method for the cancel button click event. This then cancels then formally cancels the token object that manages cancellation, stops and disables the timer, and updates the form UI Thread GUI.

[insert code here]

#### Labels and UI Thread Synchronization

When the client program runs in a test, it continuously updates the money pot totals and the test time remaining. These are shown as labels on the client GUI windows form.

Since the client work is done primarily on new thread that runs in parallel, anytime the GUI needs to be updated, there must be cross-thread synchronization. The static and instance money pot values are shown as labels on the GUI with the test time countdown remaining. The service call work needs to update these three labels and does so in a special method called UpdateLabelText(). The method receives two input parameters: Label control object and a message (string) to replace the label.

First the method checks if cross-thread synchronization is required by checking the specific label control’s *InvokeRequired* property. So, if this method is called from our task, the test for InvokeRequired required will return *true*. This will then *asynchronously* call this method *recursively*. This time, however, since it was invoked upon the UI thread, InvokeRequired will be false so the else portion of the function executes, updating the Label.

[insert code here]

### ChannelFactory Proxy and Application Configuration

The application configuration file (App.config) specifies the service endpoint address, binding, and contract information for the client to create a proxy and connect. The service contract information is in the Shared Class Library that has the service interface. The code is available on GitHub here [].

[insert code here]

## Demo

## Code

The entire project code repository is available on GitHub here.