WCF: Duplex Operations and UI Threads

jeff.barnes, 20 Feb 2007 CPOL

★★★★★ 4.91 (155 votes)

This article examines how duplex operations are implemented in WCF as well as some of the issues that may arise when dealing with UI threads.

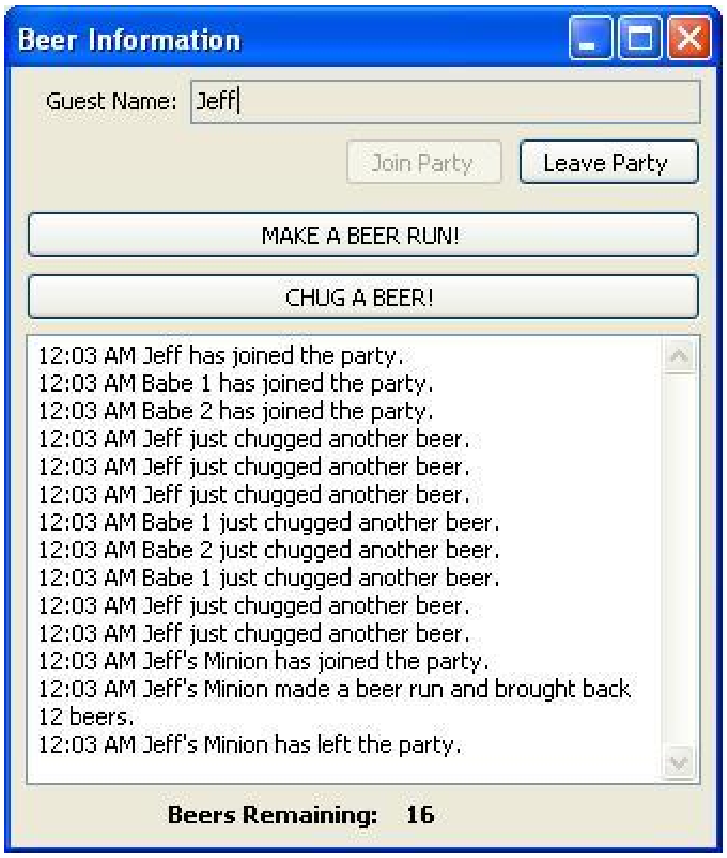
Download source files - 25.6 KB

Introduction

Simply put, duplex operations provide a mechanism for allowing a service to callback to the client. When a contract is defined for a service, it is possible to specify a corresponding callback contract. The standard service contract defines the operations of the service that can be invoked by the client. The callback contract defines the operations of the client that can be invoked by the service. It is the client's responsibility to implement the callback contract and host the callback object. Each time the client invokes a service operation that is associated to the callback contract, the client supplies the necessary information for the service to locate and invoke the callback operation on the client.

One of the most popular uses for duplex operations seems to be for events. This is typically accomplished by using a publish-subscribe pattern. One or more clients will subscribe to the service. When something of interest occurs, the service will publish the information to the subscribed clients.

This is the approach used by the demo. It is based on the concept of assisting the host of a party by tracking guests and their consumption of beer. Although intentionally humorous, it effectively illustrates the key concepts involved with duplex operations. Each guest subscribes to the service by joining the party. When something interesting happens, such as other guests joining/leaving the party as well as the consumption of beer, the service publishes the information to the subscribed clients.



Jeff

```
12:03 AM Jeff has joined the party.
12:03 AM Babe 1 has joined the party.
12:03 AM Babe 2 has joined the party.
12:03 AM Jeff just chugged another beer.
12:03 AM Jeff just chugged another beer.
12:03 AM Jeff just chugged another beer.
12:03 AM Babe 1 just chugged another beer.
12:03 AM Babe 2 just chugged another beer.
12:03 AM Babe 1 just chugged another beer.
12:03 AM Jeff just chugged another beer.
12:03 AM Jeff just chugged another beer.
12:03 AM Jeff's Minion has joined the party.
12:03 AM Jeff's Minion made a beer run and brought back
12 beers.
12:03 AM Jeff's Minion has left the party.
```

Beers Remaining: 16

Let's take a closer look at how the magic happens.

Service Contract and Callback Contract

As usual, an interface is decorated with the `ServiceContract` attribute to inform the WCF runtime that it contains the operations of a service. The`CallbackContract` property of the `ServiceContract` attribute is used to specify the interface that defines the callback operations. This creates an association between the two interfaces. In order to consume the service operations, the client must implement the callback contract and host the object for invocation by the service.

```
[ServiceContract(
    SessionMode = SessionMode.Required,
    CallbackContract = typeof(IBeerInventoryCallback))]
public interface IBeerInventory
{
    [OperationContract()]
    int JoinTheParty(string guestName);

    [OperationContract(IsOneWay = true)]
    void MakeBeerRun(string guestName, int numberOfBeers);

    [OperationContract(IsOneWay = true)]
    void DrinkBeer(string guestName);

    [OperationContract(IsOneWay = true)]
    void LeaveTheParty(string guestName);
}

public interface IBeerInventoryCallback
{
    [OperationContract(IsOneWay = true)]
    void NotifyGuestJoinedParty(string guestName);

    [OperationContract(IsOneWay = true)]
    void NotifyBeerInventoryChanged(string guestName, int numberOfBeers);

    [OperationContract(IsOneWay = true)]
    void NotifyGuestLeftParty(string guestName);
}
```

It should be noted that the `IBeerInventoryCallback` interface is not decorated with the `ServiceContract` attribute. Since it was specified as the callback contract, the WCF runtime will implicitly add a `ServiceContract` attribute to the interface. However, you can still explicitly add it if you so choose.

Service Implementation

In order to invoke the client callback from the service, it is necessary to acquire a reference to the callback object. Each time the client invokes a service operation associated to the callback contract, it supplies a callback channel that can be used to communicate with the callback object. The callback channel can be found within the `OperationContext` as shown below:

```
private List<IBeerInventoryCallback> _callbackList;

public int JoinTheParty(string guestName)
{
   IBeerInventoryCallback guest =
       OperationContext.Current.GetCallbackChannel<IBeerInventoryCallback>();

    if (!_callbackList.Contains(guest))
    {
        _callbackList.Add(guest);
    }

    _callbackList.ForEach(
        delegate(IBeerInventoryCallback callback)
        { callback.NotifyGuestJoinedParty(guestName); });

    return _beerInventory;
}
```

The service stores a reference to the callback channel in a collection. When the service is ready to publish a notification to subscribers, each callback channel in the collection is invoked. Now, this is where you can run into some issues if your service isn't properly configured and/or implemented. Specifically, the concurrency model becomes a factor.

Concurrency Mode

By default, WCF services are configured to be single-threaded. This means that only one thread will process messages at any given time for a service instance. Consequently, if additional messages arrive while a message is already being processed, they are blocked until the current message has completed processing and releases the lock.

If you attempt to invoke a client callback during the middle of a service operation that is single-threaded, a nasty little scenario known as a deadlock will occur. This is due to the locking required by the concurrency model. When the callback operation is invoked on the client, the service must wait for the reply to return for processing. However, the service is already locked for the processing of the current operation. So, a deadlock occurs. Fortunately, WCF can detect when you attempt to do this sort of thing and throws an `InvalidOperationException`, which is better than a deadlock.

So, how do you invoke a callback in the middle of a service operation? It boils down to pretty much three choices:

1. In the `ServiceBehavior` attribute, set the `ConcurrencyMode` property to`Reentrant`. This will still use a single-threaded model, but WCF will release the lock when you invoke a callback on a client. This allows the service instance to process the reply from the callback. However, you must ensure the state of any local data is valid before and after the invocation of the callback.
2. In the `ServiceBehavior` attribute, set the `ConcurrencyMode` property to`Multiple`. This enables a multi-threaded model, which requires the service implementation to handle the necessary locking.
3. In the `OperationContract` attribute of the callback methods, set the`IsOneWay` property to true. This allows WCF to invoke the callback operation on the client without requiring a lock because there will not be any reply to process.

The demo uses the third option.

Client Implementation

As previously mentioned, the client must implement the callback contract and host the callback object. For the demo, the callback contract has been directly implemented in the client form. When the form is loaded, an `InstanceContext` is instantiated and supplied to the service proxy. The `InstanceContext` is a WCF object that handles the necessary infrastructure to host the callback object, and it significantly simplifies the amount of required code. Once the `InstanceContext`and service proxy have been instantiated, a connection can be opened with the service for invocation of operations.

```
_proxy = new BeerInventoryServiceClient(new InstanceContext(this));
_proxy.Open();
```

UI Thread Woes

Now, when the user interacts with the form, the corresponding operations are invoked on the service. Some of these operations result in a callback to the subscribed clients. However, this poses a problem that is specific to Windows form (and WPF) applications. If the UI thread invokes a service operation that results in a callback operation, a deadlock scenario will occur.

Consider the following line of code that is executed when the Join Party button is clicked:

```
this.BeerInventory = _proxy.JoinTheParty(this.txtGuestName.Text);
```

This may seem innocent at first, but consider what is taking place. When the service operation is invoked from the UI thread, it will block until the return value has been received. However, the service operation invokes a callback to the client prior to the sending the return value. The callback operation will be marshaled to the UI thread. Since the UI thread is still waiting for the return value, a deadlock will occur. The root of this problem has to do with synchronization contexts.

By default, WCF provides thread affinity to the current synchronization context that is used to establish a connection with a service. All service requests and replies will be executed on the thread of the originating synchronization context. In some situations, this may be desired behavior. At other times, it may not. In our case, it is causing a problem.

Synchronization Contexts

Fortunately, WCF provides a simple mechanism for overriding the automatic association of synchronization contexts. This behavior can be turned off by setting the `UseSynchronizationContext` property of the `CallbackBehavior`attribute to `false`. In doing so, WCF will no longer guarantee a particular thread to be responsible for processing service requests. Instead, the operations will be delegated to worker threads.

```
[CallbackBehavior(UseSynchronizationContext = false)]
```

Unfortunately, this solves one problem and creates another for our demo. The callback operations on the client require updating attributes of the UI. But, only the UI thread can directly manipulate the properties of form controls. Since the callback operations will be running on a worker thread, it requires a little extra effort to manipulate the UI on the UI thread. There are a variety of ways to accomplish the task, but the demo employs the use of the `SendOrPostCallback`.

`SendOrPostCallback` is a special delegate for dispatching messages to a specific synchronization context. When the form is loaded, a reference is captured to the UI synchronization context. In the callback implementation, a `SendOrPostCallback`delegate is created for the necessary UI updates. The delegate is then invoked on the UI synchronization context to avoid security errors. This provides a mechanism for the worker thread executing the callback to communicate with the UI thread for the purpose of updating controls on the form.

```
SendOrPostCallback callback =
    delegate (object state)
    { this.WritePartyLogMessage(String.Format("{0} has joined the party.",
    state.ToString())); };

_uiSyncContext.Post(callback, guestName);
```

Conclusion

Duplex operations are a very powerful concept that is easily implemented in WCF. Using this ability, it is very easy to achieve event-like behavior between a service and client. Hopefully, this article has provided you with enough information to get started applying the concept.

License

Share

About the Author

jeff.barnes

Software Developer (Senior)

United States 🇺🇸

I'm a passionate software developer and advocate of the Microsoft .NET platform. In my opinion, software development is a craft that necessitates a conscious effort to continually improve your skills rather than falling into the trap of complacency. I was also awarded as a Microsoft MVP in Connected Systems in 2008, 2009, and 2010.