Introduction

Tuesday, January 27, 2015 1:09 PM

Overview

FabAct is an Actor programming model for Windows Fabric. Windows Fabric is a platform for building highly reliable, scalable applications for both cloud and on premise that are easy to develop and manage.

FabAct provides an asynchronous, single-threaded actor model. The actors represents the unit of state and computation that are distributed throughout the cluster to achieve high scalability. When the actor processes or the nodes on which they are hosted fails, the system recreates them on different node or processes. The framework leverages the distributed store provided by underlying Windows Fabric platform to provide highly available and consistent state management for the application developers. This makes developing and maintaining distributed cloud applications extremely easy.

Actors

The Actors are isolated single-threaded components that encapsulate both state and behavior. They are similar to .NET objects and hence provides a natural programming model to the developers. Actors interacts with rest of the system, including other actors by passing asynchronous messages with request-response pattern. These interactions are defined in an interface as asynchronous methods. Actors implementing these methods do not have to deal with locking or other concurrency issues as the framework provides a turn based concurrency for the actor methods. This means that not more than one thread will be active at any time inside the actor methods.

The actors in FabAct are virtual actors meaning that they always exists. You do not need to explicitly create them nor destroy them. The framework along with the underlying platform provides the location transparency and failover for the actors. If an actor is not used for certain time the framework will garbage collect it and will recreate it at later time if required.

Actors and Windows Fabric Services

Every Actor type is mapped to a Windows Fabric Service which is partitioned based on the int64 keys. The number of partitions for a service can be specified in the ApplicationManifest.xml generated by FabActUtil.exe. Each partition of the service can have one or more replica depending upon the state provider used as well as settings in the ApplicationManifest.xml. The platform places the replicas of various partition on different nodes available in the cluster. These partitions run inside the Code that hosts the service and hence the Actors. This is the executable that registers the Actor implementations with Windows Fabric runtime.

The partitions provides automatic scaling for your applications whereas the replicas provides reliability for the state. As you add more nodes the resource manager of Windows Fabric will spread out the partition to utilize the cluster resources more effectively. The resource manager optimizes the placement based on the available fault and upgrade domains in the cluster. Note that two replicas of the same partition are never placed in on the same node.

Concurrency

Thursday, January 29, 2015 1:31 PM

Actor framework provides a simple turn based concurrency for actor methods. This means that no more than one thread can be active inside the actor method any time. The framework can provide this guarantees for the methods invocations that are done in response to receiving a message, timer callback or reminder callback. If the actor code creates a Task that is not associated with the Task returned by the actor methods or it creates a thread, the framework cannot provide any guarantees with respect to the concurrency of those executions. To perform a background operation, please use Actor Timers or Actor Reminders that respect the turn based concurrency.

Please note that a turn consistent complete execution of an actor method in response to the request from other actors or clients, or a timer / reminder callback. Even though these methods and callback are asynchronous. there is no interleaving. A turn must be completed fully before a new call is allowed.

The framework however, allows reentrancy by default. This means that if an actor method of Actor A calls method on Actor B which in turn calls another method on actor A, that method is allowed to run as it is part of the same logical call chain context. All timer and reminder calls start with the new logical call context. See Reentrancy section for more details.

Reentrancy

Thursday, January 29, 2015 6:29 PM

FabAct by default allows logical call context based reentrancy. This allows for actors to be reentrant if they are in the same call context chain. For example if actor A sends message to Actor B who sends message to Actor C. As part of the message processing if actor C calls actor A, the message is reentrant so will be allowed. Any other messages that are part of different call context will be blocked on actor A till it completes processing.

Actors that want to disallow logical call context based reentrancy can disable it by decorating the actor class with ReentranAttribute(ReentrancyMode.Disallowed).

```
public enum ReentrancyMode
{
    LogicalCallContext = 1,
    Disallowed = 2
}
```

The following code shows actor class that sets the reentrancy mode to Disallowed. In this case if an actor sends a reentrant message to another actor an exception of type FabricException will be thrown

```
[Reentrant(ReentrancyMode.Disallowed)]
class VoicemailBoxActor : Actor<VoicemailBox>, IVoicemailBoxActor
{

    ...
}
```

State Management

Thursday, January 15, 2015 2:18 PM

FabAct allows you to create the actors that are either stateless or stateful.

Stateless Actors

Stateless actors, as the name suggests do not have the state that is managed by the framework. The statekess actors derive from Actor base class. They can have member variables and those would be preserved throughout the lifetime of that actor just like any other .NET types. However if that actor instance is garbage collected after being idle for some time it would lose the state stored in its member variables. Similarly if the actor process or the node fails it would lose the state stored in its member variables as well.

Following is an example of a stateless actor.

```
class HelloActor : Actor, IHello
{
    public Task<string> SayHello(string greeting)
    {
        return Task.FromResult("You said: '" + greeting + "', I say: Hello Actors!");
    }
}
```

Stateful Actors

Stateful actors have a state that needs to be preserved across activation and failovers. The stateful actors derive from Actor<TState> base class where TState is the type of the state that needs to be preserved across activations and failovers. The state can be accessed in the actor methods via State property on the base class Actor<TState>. Following is an example of a stateful actor accessing the state.

```
class VoicemailBoxActor : Actor<VoicemailBox>, IVoicemailBoxActor
{
    public Task<List<Voicemail>> GetMessagesAsync()
    {
        return Task.FromResult(State.MessageList);
    }

    ...
}
```

The storage and retrieval of the state is provided by an actor state provider. There are few default actor state providers that are included in the framework. The durability and reliability of the state is determined by the guarantees offered by the state provider. When an actor is activated its State is loaded in memory. Once the actor method returns modified state is automatically saved by the framework by calling a method on the state provider. If failure occurs during the save state, the framework recycles that actor instance. A new actor instance is created and loaded with the last consistent data from the state provider.

State provider can be configured per actor or for all actors within an assembly by the state provider specific attribute. By default a stateful actor uses key value store actor state provider. This state provider is built on the distributed Key-Value store provided by Windows Fabric platform. The state is durably saved on the local disk as well as replicated and durably saved on secondary replicas as well. The state save is complete only when a quorum of replicas has committed the state to their local disks. The Key-Value store has advance capabilities to detect inconsistencies such as false progress and correct them automatically. More details on the KvsActorStateProvider can be found here.

The framework also includes a VolatileActorStateProvider. This state provider replicates the state to replicas but the state remains in-memory on the replica. If one replica goes down and comes back up, its state is rebuilt from the other replica. However if all of the replicas (copies of the state) go down simultaneously the state data will be lost. This state provider is

suitable for applications where the data can survive failures of few replicas and can survive the planned failovers such as upgrades, but can be recreated if all replicas (copies) are lost. You can configure your stateful actor to use volatile actor state provider by adding the following attribute to the actor class or an assembly level attribute.

The following code snippet shows how to changes all actors in the assembly that does not have an explicit state provider attribute to use VolatileActorStateProvider.

```
[assembly:System.Fabric.Services.Actor.VolatileActorStateProvider]
```

The following code snippet shows how to change the state provider for a particular actor, VoicemailBox in this case to be VolatileActorStateProvider.

```
[VolatileActorStateProvider]
public class VoicemailBoxActor : Actor<VoicemailBox>, IVoicemailBoxActor
{
    public Task<List<Voicemail>> GetMessagesAsync()
    {
        return Task.FromResult(State.MessageList);
    }

    ...
}
```

Please note that changing the state provider requires the actor service to be recreated. State providers cannot be changed as part of the application upgrade.

Readonly Methods

By default the framework saves Actor state upon exiting from an actor method call, timer callback and reminder callback. No other actor call is allowed until the save state is complete. Depending upon the state provider, saving state may take time and during this time no other actor methods are being allowed in the actor. For example, the default KvsActorStateProvider replicates the data to a replica set and only when a quorum of replicas have committed the data to a persisted store, the save state is completed.

There may be actor method that do not modify the state, in that case the additional time spent on saving the state can affect the overall throughput of the system. To avoid that you can mark the methods and timer callback that do not modify the state as Readonly.

The example below shows how to mark an actor method as readonly using `Readonly` attribute.

```
public interface IVoicemailBoxActor : IActor
{
    [Readonly]
    Task<List<Voicemail>> GetMessagesAsync();
```

Similarly you can pass Readonly flag when registering a timer or reminder.

Communication

Tuesday, January 27, 2015 4:14 PM

The actor framework provides communication between the actor instance and the actor client. The framework also provides the location transparency and failover. The ActorProxy class on the client side perform the necessary resolution and locates the actor service partition where the actor with the specified id is hosted and opens a communication channel with it. The ActorProxy retries on the communication failures and in case of failovers. This does mean that it is possible for an Actor implementation to get duplicate messages from the same client. ActorProxy should be used for client to Actor as well as Actor to Actor communication.

The code snippet below shows the example of creating the ActorProxy to communicate to the actor that implements IHelloWorld actor interface.

```
var friend = ActorProxy.Create<IHello>(ActorId.GetRandom(), ApplicationName);

Console.WriteLine("\n\nFrom Actor {1}: {0}\n\n", friend.SayHello("Good morning!").Result,
friend.GetActorId());
```

Lifecycle

Tuesday, January 27, 2015 3:57 PM

An Actor is activated when first call is made to it and its deactivated(Garbage Collected by framework) if it's not used for some period of time. To configure this time period please see section on Actor Garbage Collection below.

What happens on Actor Activation?

- When a call comes for an actor and it's not already active, a new actor is created.
- Its state is loaded (if it's a stateful actor)
- `OnActivateAsync` method (which can be overridden in actor implementation) is called.
- Its added to an Active Actors table.

What happens on Actor Deactivation?

- When an actor is not used for some period of time, its removed from the table of Active Actors. `OnDeactivateAsync` method (which can be overridden in actor implementation) is called which clears all the timers for the actor.

Actor Garbage Collection

The framework periodically scans for actors that have not been used for some period of time, and deactivates them. Once they are deactivated they can be garbage collected by CLR.

What counts as "being used" for the purpose of garbage collection?

- Receiving a call.
- Activation of a Reminder

By default the framework scans the actors every 60 seconds and collects the actors that are not used for more than 60 minutes. So if an Actor was activated at time T1, framework will scan every 60 seconds to see if Actor can be garbage collected and after T1 + 3600 seconds it will be collected. Now if the Actor gets reused at T1 + 3000 seconds, then framework will wait again for 3600 seconds before collecting it, scanning every 60 seconds. Please note that an Actor will never get garbage collected while it is executing one of its method, no matter how much time is spent in the method call. The idle timeout countdown starts only after the actor is truly idle, i.e. it is not executing any of the actor methods, actor timer callbacks or actor reminder callbacks.

While the actor will not get garbage collected during the execution of the timer callback, the actor timers will not activate the actor nor it can keep the actor alive (keep it from being garbage collected). See the description at the end of the example below.

Typically you do not need to change these settings. These intervals can be changed at an assembly level for all actor types in that assembly or at an actor type level by `ActorGarbageCollection` attribute.

The example below shows the change in the garbage collection intervals for `HelloActor`.

```
[ActorGarbageCollection(IdleTimeoutInSeconds = 10, ScanIntervalInSeconds = 2)]
class HelloActor : Actor, Ihello
{
    public Task<string> SayHello(string greeting)
    {
        return Task.FromResult("You said: '" + greeting + "', I say: Hello Actors!");
    }
}
```

The framework scans for actors every `ScanIntervalInSeconds` to see if it can be garbage collected and collects it if it's not being used for `IdleTimeoutInSeconds`. If Actor gets reused, then idle time for Actor is reset to 0.

In the example above, if an Actor was activated at time T1, framework will scan every 2 seconds to see if Actor can be garbage collected and after T1+10 it will be collected, if the actor does not get reused. Now if the Actor gets reused at T1+3 seconds,

Timers

Thursday, January 29, 2015 1:29 PM

Actor timers provides a simple wrapper around .NET timer such that the callback methods respects the turn based concurrency guarantees provided by the actor framework.

The example below shows the use of timer APIs. The APIs are very similar to the .NET timer. In the example below when the timer is due `MoveObject` method will be called by the framework and it is guaranteed to respect the turn based concurrency, which means that no other actor methods or timer callbacks will be in progress when this callback is invoked.

The next period of the timer starts after the callback returns. The framework will also try to save the state when the method returns if the Actor is a stateful actor like in this case below. If an error occurs in saving the state, that actor object will be deactivated and a new instance will be activated. A callback method that does not modify the actor state can be registered as a readonly timer callback in RegisterTimer.

```
class VisualObjectActor : Actor<VisualObject>, IVisualObject
{
    private IActorTimer _updateTimer;

    public override Task OnActivateAsync()
    {
  ...

        _updateTimer = RegisterTimer(
            MoveObject,                     // callback method
            null,                           // state to be passed to the callback method
            TimeSpan.FromMilliseconds(15),  // amount of time to delay before callback is invoked
            TimeSpan.FromMilliseconds(15)); // time interval between invocation of the callback method

        return base.OnActivateAsync();
    }


    public override Task OnDeactivateAsync()
    {
        if (_updateTimer != null)
        {
            UnregisterTimer(_updateTimer);
        }

        return base.OnDeactivateAsync();
    }

    private Task MoveObject(object state)
    {

   ...
        return TaskDone.Done;
    }
}
```

Reminders

Thursday, January 29, 2015 5:55 PM

FabAct provides reminders are a mechanism to trigger persistent callbacks on Actor. Reminders unlike Timers are triggered under all circumstances until the Reminder is explicitly unregistered by the Actor. Actors that need to provide support for reminders must implement IRemindable interface

```
public interface IRemindable
{
    Task ReceiveReminderAsync(string reminderName, byte[] context, TimeSpan dueTime, TimeSpan
period);
}
```

When a reminder is triggered, FabAct runtime will invoke ReceiveReminderAsync method on the Actor and pass in the context duetime and period parameters specified during registration. To register a reminder an actor can call Register method provided on base class

```
async Task<IActorReminder> RegisterReminder(string reminderName, byte[] context, TimeSpan dueTime,
TimeSpan period, ActorReminderAttributes attribute);
```

ReminderName parameter is a string that uniquely identifies the reminder for an Actor. Context contains any state that must be passed to ReceiveReminderAsync method. DueTime is the time span after which the first reminder fires and period is the time span for repeated reminder invocations.

ActorReminderAttributes specify if state must be saved after ReceiveReminderAsync method call returns from Actor. The attribute can have the following values

```
public enum ActorReminderAttributes : long
{
    None = 0x00,
    ReadOnly = 0x01
}
```

A registered reminder will keep triggering for an Actor even after an actor has been garbage collected. To unregister a reminder Unregister method should be called.

```
async Task UnregisterReminder(IActorReminder reminder);
```

Events

Monday, February 2, 2015 11:29 AM

Actor events provides a way to send best effort notifications of changes from the Actor to the clients. Actor events are designed for Actor-Client communication and should NOT be used for Actor-to-Actor communication.

Following code snippets shows how to use actor events in your application.

Define an interface that describes the events published by the actor. The arguments of the methods must be data contract serializable. The methods must return void as event notifications are one-way and best effort.

```
  public interface IGameEvents
  {
     void GameScoreUpdated(Guid gameId, string currentScore);
  }
```

Declare the events published by the actor in the actor interface.

```
public interface IGameActor : IActor, IActorEventPublisher<IGameEvents>
{
    Task UpdateGameStatus(GameStatus status);

    [Readonly]
    Task<string> GetGameScore();
}
```

On the client side implement the event handler.

```
class GameEventsHandler : IGameEvents
{
    public void GameScoreUpdated(Guid gameId, string currentScore)
    {
        Console.WriteLine(@"Updates: Game: {0}, Score: {1}", gameId, currentScore);
    }
}
```

On the client, create proxy to the actor that publishes the event and subscribe to them.

```
var proxy = ActorProxy.Create<IGameActor>(
              new ActorId(Guid.Parse(arg)), ApplicationName);

proxy.SubscribeAsync(new GameEventsHandler()).Wait();
```

In the event of failovers the actor may failover to a different process or node. The actor proxy manages the active subscriptions and automatically re-subscribes them. You can control the re-subscription interval

through SubscribeAsync API. To unsubscribe use Unsubscribe API.

On the actor simply publishes the events as they happen. If there are subscribes subscribed to the event, the framework will send them the notification.

```
var ev = GetEvent<IGameEvents>();
ev.GameScoreUpdated(Id.GetGuidId(), State.Status.Score);
```