Windows Communication Foundation

Windows Communication Foundation (WCF), released in .NET 3.0, is quickly becoming the de-facto standard for most networking needs in .NET applications. It offers an unparalleled choice of network protocols and customizations and is continuously being extended with new .NET releases. This section covers WCF performance optimizations.

Throttling

WCF, especially before .NET Framework 4.0, has conservative throttling values by default. These are designed to protect against Denial of Service (DoS) attacks, but, unfortunately in the real world, they are often set too low to be useful.

Throttling settings can be modified by editing the `system.serviceModel` section in either app.config (for desktop applications) or web.config for ASP.NET applications:

```
<system.serviceModel>
  <behaviors>
    <serviceBehaviors>
     <behavior>
      <serviceThrottling>
       <serviceThrottling maxConcurrentCalls = "16"
        maxConcurrentSessions = "10" maxConcurrentInstances = "26" />
```

Another way to change these parameters is by setting properties on a `ServiceThrottling` object during service creation time:

```
Uri baseAddress = new Uri("http://localhost:8001/Simple
");
ServiceHost serviceHost = new ServiceHost(typeof(CalculatorService), baseAddress);

serviceHost.AddServiceEndpoint(
  typeof(ICalculator),
  new WSHttpBinding(),
  "CalculatorServiceObject");

serviceHost.Open();

IChannelListener icl = serviceHost.ChannelDispatchers[0].Listener;
ChannelDispatcher dispatcher = new ChannelDispatcher(icl);
ServiceThrottle throttle = dispatcher.ServiceThrottle;

throttle.MaxConcurrentSessions = 10;
throttle.MaxConcurrentCalls = 16;
throttle.MaxConcurrentInstances = 26;
```

Let us understand what these parameters mean.

- `maxConcurrentSessions` limits the number of messages that currently process across a `ServiceHost`. Calls in excess of the limit are queued. Default value is 10 for .NET 3.5 and 100 times the number of processors in .NET 4.
- `maxConcurrentCalls` limits the number of `InstanceContext` objects that execute at one time across a `ServiceHost`. Requests to create additional instances are queued and complete when a slot below the limit becomes available.
- Default value is 16 for .NET 3.5 and 16 times the number of processors in .NET 4.
- `maxConcurrentInstances` limits the number of sessions a `ServiceHost` object can accept. The service accepts connections in excess of the limit, but only the channels below the limit are active (messages are read from the channel).
- Default value is 26 for .NET 3.5 and 116 times the number of processors in .NET 4.

Another important limit is the number of concurrent connections an application is allowed per host, which is two by default. If your ASP.NET application calls an external WCF service, this limit may be a significant bottleneck. This is a configuration example that sets these limits:

```
<system.net>
    <connectionManagement>
     <add address = "*" maxconnection = "100" />
  </connectionManagement>
</system.net>
```

Process Model

When writing a WCF service, you need to determine its activation and concurrency model. This is controlled by `ServiceBehavior` attribute's `InstanceContextMode` and `ConcurrencyMode` properties, respectively. The meaning of `InstanceContextMode` values is as follows:

- `PerCall` – A service object instance is created for each call.
- `PerSession` (default) – A service object instance is created for each session. If the channel does not support sessions, this behaves like `PerCall`.
- `Single` – A single service instance is re-used for all calls.

The meaning of `ConcurrencyMode` values is as follows:

- `Single` (default) – The service object is single-threaded and does not support re-entrancy. If `InstanceContextMode` is set to `Single` and it already services a request, then additional requests have to wait for their turn.
- `Reentrant` – The service object is single-threaded but is re-entrant. If the service calls another service, it may be re-entered. It is your responsibility to ensure object state is left in a consistent state prior to the invocation of another service.
- `Multiple` – No synchronization guarantees are made, and the service must handle synchronization on its own to ensure consistency of state.

Do not use `Single` or `Reentrant ConcurrencyMode` in conjunction with `Single InstanceContextMode`. If you use `Multiple ConcurrencyMode`, use fine-grained locking so that better concurrency is achieved.

WCF invokes your service objects from .NET thread pool I/O completion threads, previously described within this chapter. If during servicing you perform synchronous I/O or do waits, you may need to increase the number of thread pool threads allowed by editing the `system.web` configuration section for an ASP.NET application (see below) or by calling `ThreadPool.SetMinThreads` and `ThreadPool.SetMaxThreads` in a desktop application.

```
<system.web>
  <processModel
  ...
  enable = "true"
  autoConfig = "false"

maxWorkerThreads ="80"
maxIoThreads = "80"
minWorkerThreads = "40"
minIoThreads = "40"

  />
```

Caching

WCF does not ship with built-in support for caching. Even if you are hosting your WCF service in IIS, it still cannot use its cache by default. To enable caching, mark your WCF service with the `AspNetCompatibilityRequirements` attribute.

```
[AspNetCompatibilityRequirements(RequirementsMode = AspNetCompatibilityRequirementsMode.Allowed)]
```

In addition, enable ASP.NET compatibility by editing web.config and adding the following element under the `system.serviceModel` section:

```
<serviceHostingEnvironment aspNetCompatibilityEnabled = "true" />
```

Starting with .NET Framework 4.0, you can use the `new System.Runtime.Caching` types to implement caching. It is not dependent on the System.Web assembly, so it is not limited to ASP.NET.

Asynchronous WCF Clients and Servers

WCF lets you issue asynchronous operation on both the client and server side. Each side can independently decide whether it operates synchronously or asynchronously.

On the client side, there are two ways to invoke a service asynchronously: event-based and .NET async pattern based. The event-based model is not compatible with channels created with `ChannelFactory`. To use the event-based model, generate a service proxy by using the svcutil.exe tool with both the `/async` and `/tcv:Version35` switches:

```
svcutil /n:http://Microsoft.ServiceModel.Samples,Microsoft.ServiceModel.Sampleshttp://localhost:8000/servicemodelsamples/service/mex /async /tcv
```

The generated proxy can then be used as follows:

```
// Asynchronous callbacks for displaying results.
static void AddCallback(object sender, AddCompletedEventArgs e) {
  Console.WriteLine("Add Result: {0}", e.Result);
}

static void Main(String[] args) {
  CalculatorClient client = new CalculatorClient();
  client.AddCompleted + = new EventHandler < AddCompletedEventArgs > (AddCallback);
  client.AddAsync(100.0, 200.0);
}
```

In the `IAsyncResult`-based model, you use svcutil to create a proxy specifying the `/async` switch but without also specifying the `/tcv:Version35` switch. You then call the `BeginXXX` methods on the proxy and provide a completion callback as follows:

```
static void AddCallback(IAsyncResult ar) {
  double result = ((CalculatorClient)ar.AsyncState).EndAdd(ar);
  Console.WriteLine("Add Result: {0}", result);
}

static void Main(String[] args) {
  ChannelFactory < ICalculatorChannel > factory = new ChannelFactory < ICalculatorChannel > ();
  ICalculatorChannel channelClient = factory.CreateChannel();
  IAsyncResult arAdd = channelClient.BeginAdd(100.0, 200.0, AddCallback, channelClient);
}
```

On the server, asynchrony is implemented by creating `BeginXX` and `EndXX` versions of your contract operations. You should not have another operation named the same but without the Begin/End prefix, because WCF will invoke it instead. Follow these naming conventions, because WCF requires it.

The `BeginXX` method should take input parameters and return `IAsyncResult` with little processing; I/O should be done asynchronously. The `BeginXX` method (and it alone) should have the `OperationContract` attribute applied with the `AsyncPattern` parameter set to `true`.

The `EndXX` method should take an `IAsyncResult`, have the desired return value, and have the desired output parameters. The `IAsyncResult` object (returned from `BeginXX`) should contain all the necessary information to return the result.

Additionally, WCF 4.5 supports the new `Task`-based `async`/`await` pattern in both server and client code. For example:

```
//Task-based asynchronous service
public class StockQuoteService : IStockQuoteService {
    async public Task<double> GetStockPrice(string stockSymbol) {
     double price = await FetchStockPriceFromDB();
     return price;
  }
}

//Task-based asynchronous client
public class TestServiceClient : ClientBase < IStockQuoteService>, IStockQuoteService {
   public Task<double> GetStockPriceAsync(string stockSymbol) {
    return Channel.GetStockPriceAsync();
  }
}
```

Bindings

When designing your WCF service, it is important to select the right binding. Each binding has its own features and performance characteristics. Choose the simplest binding, and use the smallest number of binding features that will serve your needs. Features, such as reliability, security, and authentication, add a great deal of overhead, so use them only when necessary.

For communication between processes on the same machine, the Named Pipe binding offers the best performance. For cross-machine two-way communication, the Net TCP binding offers the best performance. However, it is not interoperable and can only work with WCF clients. It is also not load-balancer-friendly, as the session becomes affine to a particular server address.

You can use a custom binary HTTP binding to get most of the performance benefits of TCP binding, while remaining compatible with load balancers. Below is an example of configuring such a binding:

```
<bindings>
  < customBinding>
    <binding name = "NetHttpBinding">
     <reliableSession />
     <compositeDuplex />
     <oneWay />
     <binaryMessageEncoding />
     <httpTransport />
     </binding>
   </customBinding>
  <basicHttpBinding>
  <binding name = "BasicMtom" messageEncoding = "Mtom" />
  </basicHttpBinding>
  <wsHttpBinding>
     <binding name = "NoSecurityBinding">
              <security mode = "None" />
     </binding>
    </wsHttpBinding>
</bindings>
<services>
  <service name =    "MyServices.CalculatorService">
             <endpoint address = " " binding = "customBinding" bindingConfiguration = "NetHttpBinding"
                        contract = "MyServices.ICalculator" />
  </service>
</services>
```

Finally, choose the basic HTTP binding over the WS-compatible one. The latter has a significantly more verbose messaging format.