SILVERKEY TECHNOLOGIES

Enterprise Service Bus (ESB) With nServiceBus

Introduction and Implementation Tutorial

Mohammad Ahmed Meligy

4/22/2008

Revision History

Author: Mohammad Meligy	Revision Number: 1.1	Project: Not specified
Status: Public Draft	Last Updated: 4/22/2008	Feature ID:
Related Document(s):		

Author: Mohammad Meligy	Revision Number: 1	Project:
Status: Draft	Last Updated: 4/19/2008	Feature ID:
Related Document(s):		

Overview

This document shows the use of nServiceBus library (www.nservicebus.com) for managing enterprise messaging using the service bus pattern. The model explained is an advancement of a publisher / subscriber model.

Context

The document is written to be guidance for developers involved in intercommunication between components in enterprise project, however, it is written with no specific relevance to the project components, and can be used for any project that implements this kind of messaging.

Glossary

There're some common terms used in this document:

- Message
  A message is the object that groups the data you want to pass from a module to another module or system for processing outside your module.
- Publisher
  This is the code that creates the message and puts it publicly to be available for later processing.
  Note: It's very common when dealing with nServiceBus to call the publisher as a "Client"
- Subscriber
  This is the code that checks for published messages and passes is responsible for performing any desired operations on them.
  Note that the subscriber that subscribes to a certain request message for example may want to send a response message in return, hence, it then acts as a publisher, and so. It's normal that the module that acts as a publisher acts also as a subscriber and vice-versa.
  Note: It's very common when dealing with nServiceBus to call the subscriber as a "Server"
- Service Bus
  This is the shared message store where all the messages are be written to by the publishers requesting processing, and then later read by the subscribers for processing. Once the messages are delivered to subscribers, they're typically removed from the message store.

Requirements

Goals

1. Enable asynchronous messaging across multiple servers.
2. Ensure reliability for the asynchronous messaging.
3. Enable a publisher / subscriber model for the messaging.
4. Enable multiple subscribers to each published message.
5. Enable a publisher to receive response messages from subscribers.

Non Goals

1. Enable canceling of already published messages.
2. Specify the sequence of message subscribers handling via configuration (still available from inside the subscriber code).

Major Features

1. Setting a server to act as service bus for multiple publishers/subscribers.
2. Setting the publishers to send messaging to a service bus asynchronously.
3. Setting multiple subscribers inside a service bus.
4. Limiting the handling of messages based on their type.

Dependencies

1. MSMQ – Acts as store for published messages
2. Spring.NET Library (On all the publisher and subscriber application – included with nServiceBus assemblies) – For initializing the messaging infrastructure
3. Log4net Library (http://logging.apache.org/log4net) - For logging purposes
4. Serialization (On all the code classes that are included in published / returned messages)

Usage

How it works

You create the message type, create a type that holds the processing of the message (called message handler), configure the publisher and create the message instance itself that it publishes to the bus, and, configure the subscriber to subscribe to the bus and map the message instance to new message handler instance.

You'll notice that the difference between the client and the server is really small. A client publishes a message to one or more primary buses. The server subscribes to the messages it has handlers for that come on the primary bus. But, for the server to be able to send back "response" messages to the client, the client also has its own bus (queue) that it subscribes to for handling response messages.

Defining the message contracts

You define the message type (*class*) normally in any VS project with certain considerations:

a. It should implements an interface *"NServiceBus.IMessage"*.
   This interface does not require you to implement any methods / properties.
a. It should be attributed as *"Serializable"*.
b. It's highly recommended that your message have an identity field of type GUID.

This message acts as data contract between the publisher and subscriber, and it should be available to both. This is the actual data that's being passed between them.

Example:

```
[Serializable]
public class RequestDataMessage : IMessage
{
    public Guid DataId;
}

[Serializable]
public class DataResponseMessage : IMessage
{
    public Guid DataId;
    public string Description;

    // other data
}
```

Writing the code to handle the messages

A handler is the business logic that does the actual processing of a message. When the subscriber receives a message, it checks which handler(s) it should call to process this message. A handler is independent from the subscriber and can be used with multiple subscribers though.

When creating the handler type (*class*), you should either

- implement the interface *"NServiceBus.IMessageHandler<MESSAGE_TYPE>"*
- Inherit from class *"NServiceBus.BaseMessageHandler<MESSAGE_TYPE>"*

MESSAGE_TYPE is the type of the message to handle.

The two options are equal in effect. You write a method "*public void Handle(MESSAGE_TYPE)*" where you write the code that actually processes the message.

Note: You can implement the interface multiple times for MESSAGE_TYPE_1, MESSAGE_TYPE_2*, etc,* and then you have to implement the methods "*public void Handle(MESSAGE_TYPE_1)*" and "*public void Handle(MESSAGE_TYPE_2)*", etc.

Example:

```
public class RequestDataMessageHandler : BaseMessageHandler<RequestDataMessage>
{
    public override void Handle(RequestDataMessage message)
    {
        Console.WriteLine("Received request {0}.", message.DataId);

        DataResponseMessage response = new DataResponseMessage();
        response.DataId = message.DataId;
        response.Description = (message.DataId.ToString("N"));

        this.Bus.Reply(response);
    }
}
```

In this example, the handler just creates a message of a different type (which it does not have to do or have to define) and sends it to the service buss.

The last line has the method "*Reply*" which accepts "*paramas*" of message(s) that act as response to the original message and sends the response message(s) back to the queue of the client (who sent the original request message).

Error Codes

To indicate that there was an error processing the message. NServiceBus promotes error codes and creating custom messages that represent the error against throwing an exception.

Here's a sample code or creating "*enum*" for wrapping error codes and returning it:

```
public class CommandMessageHandler : BaseMessageHandler<Command>
{
    public override void Handle(Command message)
    {
        if (message.Id % 2 == 0)
            this.Bus.Return((int)ErrorCodes.Fail);
        else
            this.Bus.Return((int)ErrorCodes.None);
    }
}
```

Setting up messaging communication

Now, you have a message type, and a message handler. What's remaining is writing the publisher that sends this message to the service bus, and the subscriber that asks the service bus to subscribe to any message of the type(s) you want, and before all that, creating the service bus itself.

Preparing the service bus

The first thing you need to do is to create the MSMQ queue that acts as a service bus. The name of the queue will later be available to publishers and subscribers to write messages to / read messages from.

Creating the publisher

The publisher is the code that sends the messages to the service bus through configurations. Before writing anything, you need to add references to the following assemblies:

1. NServiceBus
2. NServiceBus.UniCast
3. NServiceBus.UniCast.Subscriptions
4. NServiceBus.UniCast.Subscriptions.Msmq
5. NServiceBus.UniCast.Transport
6. NServiceBus.UniCast.Transport.Msmq
7. utils

The way nServiceBus highly promotes is setting the publisher configuration in the app.config/web.config of the Visual Studio project that contains publishing code using Spring.Net configuration.

A typical configuration looks like this:

1. A configuration section that registers the Spring.Net configurations

```
<configSections>
  <sectionGroup name="spring">
    <section name="context" type="Spring.Context.Support.ContextHandler, Spring.Core"/>
    <section name="objects" type="Spring.Context.Support.DefaultSectionHandler, Spring.Core" />
  </sectionGroup>
  <sectionGroup name="common">
    <section name="logging" type="Common.Logging.ConfigurationSectionHandler, Common.Logging" />
  </sectionGroup>
  <section name="log4net" type="log4net.Config.Log4NetConfigurationSectionHandler,log4net" />
</configSections>
```

   The "Common" and "log4net" parts are optional and are only used in case you want to use logging.

2. Create the standard sections for Spring.NET

```
<spring>

  <context>
    <resource uri="config://spring/objects"/>
  </context>

  <objects xmlns="http://www.springframework.net" >

    <object id="Builder" type="ObjectBuilder.SpringFramework.Builder, ObjectBuilder.SpringFramework" />

    <!-- All the required configuration for nServiceBus comes here -->


  </objects>

</spring>
```

3. Replace the part "*<!-- All the required configuration for nServiceBus comes here -->*" with the actual configuration to use. A typical configuration looks like:

```
<object id="Transport" type="NServiceBus.Unicast.Transport.Msmq.MsmqTransport, NServiceBus.Unicast.Transport.Msmq">
  <property name="InputQueue" value="client" />
  <property name="NumberOfWorkerThreads" value="1" />
  <property name="UseXmlSerialization" value="false" />
  <property name="IsTransactional" value="false" />
  <property name="PurgeOnStartup" value="false" />
</object>

<object id="Bus" type="NServiceBus.Unicast.UnicastBus, NServiceBus.Unicast">
  <property name="Builder" ref="Builder" />
  <property name="Transport" ref="Transport" />
  <property name="ImpersonateSender" value="false" />

  <property name="MessageOwners">
    <dictionary>
      <entry key="Messages" value="messagebus"/>
    </dictionary>
  </property>

  <property name="MessageHandlerAssemblies">
    <list>
    </list>
  </property>

</object>
```

The properties that matter in this context are:

- "*Transport*"
  - "*InputQueue*"

    Sets the path to the queue where the messages will be read from. This is essential as the publisher might also subscribe to messages (typically responses to published requests).

    Note: For MSMQ v3 (Windows XP, 2003), this must be a local queue.
- "*Message Owners*"

  This connects the message - based on the message type (class, the one that implements "*IMessage*" interface) – to the MSMQ address where it will be published.

  As shown above, this is written as a dictionary, where:
  - "*key*": is either the name of the message type (class) – or – the name of an assembly, so that nServiceBus looks inside this assembly and locate all the classes that implement the "*IMessage*" interface.
  - "*value*": is name of the MSMQ queue to publish the "*key*" messages to

4. Write the actual code that sends the message to the service bus

```
ObjectBuilder.SpringFramework.Builder builder = new ObjectBuilder.SpringFramework.Builder();

IBus bClient = builder.Build<IBus>();

bClient.Start();

RequestDataMessage m = new RequestDataMessage();
m.DataId = Guid.NewGuid();

bClient.Send(m);
```

In this sample, "*RequestDataMessage*" is just the message type (class).

The code is pretty simple, you:

- Use Spring.NET to create the service buss
- Create the message itself normally and fill its members
- Send the message to be published to the bus (This has some alternatives discussed later)

5. As the message is published asynchronously, you can optionally create a callback method that handles the message. A code for something like that would look like:

```
RequestDataMessage m = new RequestDataMessage();
m.DataId = Guid.NewGuid();

//notice that we're passing the message (m) as our state object
bClient.Send(m).Register(RequestDataComplete, m);
```

You send the message and sending returns an "*ICallback*" object. This object has single method "*Register*" which you use to register "*AsyncCallback*" delegate to a method that will be called asynchronously when the sever returns a reply, which is "*RequestDataComplete*" in our example. A sample implementation of this method would look like:

```
private static void RequestDataComplete(IAsyncResult asyncResult)
{
    CompletionResult result = asyncResult.AsyncState as CompletionResult;

    if (result == null)
        return;
    if (result.messages == null)
        return;
    if (result.messages.Length == 0)
        return;
    if (result.state == null)
        return;

    RequestDataMessage request = result.state as RequestDataMessage;
    if (request == null)
        return;

    DataResponseMessage response = result.messages[0] as DataResponseMessage;
    if (response == null)
        return;

    System.Diagnostics.Debug.Assert(request.DataId == response.DataId);
    Console.WriteLine("Response received with description: {0}", response.Description);
}
```

Clearly, you can see that result returned is a "*NServiceBus.CompletionResult*" which has both "*state*" which has the original sent message, and "*messages*" which has the response messages.

Creating the subscriber

The subscriber is the code that reads the messages published to the service bus, and pass the messages to the messages handler(s) set for it.

The configuration for the subscriber is very similar to the publisher:

1. Steps 1 to 3 in the publisher are the same for the subscriber
2. The final nServiceBus specific configurations would look like:

```
<object id="Transport" type="NServiceBus.Unicast.Transport.Msmq.MsmqTransport, NServiceBus.Unicast.Transport.Msmq">
  <property name="InputQueue" value="messagebus" />
  <property name="UseXmlSerialization" value="false" />
  <property name="NumberOfWorkerThreads" value="1" />
  <property name="IsTransactional" value="true" />
  <property name="PurgeOnStartup" value="false" />
  <property name="ErrorQueue" value="error" />
</object>

<object id="SubscriptionStorage"
      type="NServiceBus.Unicast.Subscriptions.Msmq.MsmqSubscriptionStorage, NServiceBus.Unicast.Subscriptions.Msmq">
  <property name="Queue" value="subscriptions" />
</object>

<object id="Bus" type="NServiceBus.Unicast.UnicastBus, NServiceBus.Unicast">
  <property name="Builder" ref="Builder" />
  <property name="Transport" ref="Transport" />
  <property name="SubscriptionStorage" ref="SubscriptionStorage" />
  <property name="ImpersonateSender" value="false" />

  <property name="MessageOwners">
    <dictionary>
      <entry key="Messages" value="messagebus"/>
    </dictionary>
  </property>

  <property name="MessageHandlerAssemblies">
    <list>
      <value>Server</value>
    </list>
  </property>

</object>

<object type="Server.RequestDataMessageHandler, Server"  singleton="false">
  <property name="Bus" ref="Bus" />
</object>
```

   Note the following sections as well:

   - "*Transport*"
     - "*InputQueue*"

       This is the MSMQ queue that the subscriber will listen to.
   - "*SubscriptionStorage*"
     - "*Queue*"

       This is the MSMQ queue where the subscription messages will be stored (The mapping between the subscriber and publishers)

       Note: For MSMQ v3 (Windows XP, 2003), this must be a local queue.
   - "*MessageHandlerAssemblies*"

     This is a list of assemblies to look in for message handler types (classes, ones that implement *"NServiceBus.IMessageHandler<MESSAGE_TYPE>"*), so that it maps the handlers to the messages it receives
3. Writing the actual code that subscribes to the bus. This is very simple piece of code

```
var builder = new ObjectBuilder.SpringFramework.Builder();
try
{
    IBus bServer = builder.Build<IBus>();
    bServer.Start();
}
catch (Exception e)
{
    //handle errors, restart the bus maybe, etc...
}
```

You can see it's very simple. You create the "*Bus*" object from the information in the config file using the "Spring" library, and just start the bus. It loads all the necessary information like which messages to subscribe to and the mapping between message types and message handlers, etc, from the config.

4. Now you need to deploy the code that runs the server. Note that the bus should be living forever! You'll want to treat it as you treat a Windows Workflow Foundation Workflow Runtime for example. Either make it an independent Windows Service, or write it in some Application even of the global.asax of a web application.

Logging (Optional)

For the error logging and such, nServiceBus library has dependency on another library called Log4net. You can use that for any kind of logging. Sample logging code looks like:

- `LogManager.GetLogger("hello").Debug("Started.");`
- `LogManager.GetLogger("hello").Fatal("Exiting", e);`

For Log4net library specific documentation check

Security Consideration

Seeing how nServiceBus works, you notice you have typically two queues, one for the server and another for the client. You also notice that you have two operations for each, read and write.

Security Constraints

To setup MSMQ, you can use public queues, but those are hard to create and are very insecure. Typically you use the default private queues. This causes few constraints on the queue operations.

- For reading from the queue (AKA subscribing to the bus), the queue has to be on the same machine as the subscriber. You cannot subscribe to a queue that is on a different machine if you are using MSMQ version 3 (like using Windows 2003).
- To write to the queue (AKA, publish to the bus), the OS process running client/publisher has to be running with a use account that has sufficient privileges on the machine that has the queue you want to write to. This means you'll typically be running the client and server on two machines in the same LAN with Active directory and sufficient user access, which is a common scenario anyway.

FAQ

1. Where to find a copy of the examples in this document or get a working example for nServiceBus in general?

   The code snippets in this document are mostly coming from the "FullDuplex" example that comes with the library source code.

   The source code can be found at: http://sourceforge.net/projects/nservicebus

   For few more examples, check the library Yahoo group at: http://tech.groups.yahoo.com/group/nservicebus

2. When I downloaded the nServiceBus source from the library SVN repository I found the configuration much different than described in this document. Any Idea?

   The next version of nServiceBus will have different configuration. It's not documented here because it's NOT a complete release yet so is still subject to change before the next release.

   For differences between the configuration of the current release and the upcoming release, check nServiceBus wiki page at: http://nservicebus.wiki.sourceforge.net/Configuration

   There's also incomplete documentation of the examples on nServiceBus wiki at http://nservicebus.wiki.sourceforge.net/Samples

3. Is there any documentation for nServiceBus?

   There are few resources:

   - nServiceBus Yahoo group: http://tech.groups.yahoo.com/group/nservicebus
   - nServiceBus Wiki http://nservicebus.wiki.sourceforge.net
   - Ayende's Blog Posts http://ayende.com/Blog/category/549.aspx
   - Udi Dahan's Blog (the creator of nServiceBus) http://udidahan.weblogs.us/category/nservicebus/

   However, most of those are either very limited or incomplete.

4. How do I write the name of a MSMQ queue that exists on a different computer?

   You write the name of the queue in a format like "*QUEUE_NAME @MACHINE_NAME*"

   MACHINE_NAME is the computer name of the machine that has the queue.

   QUEUE_NAME is the name of the queue itself