Implementing the Scatter Gather ESB Pattern with Neuron Pipelines

Marty Wasznicky

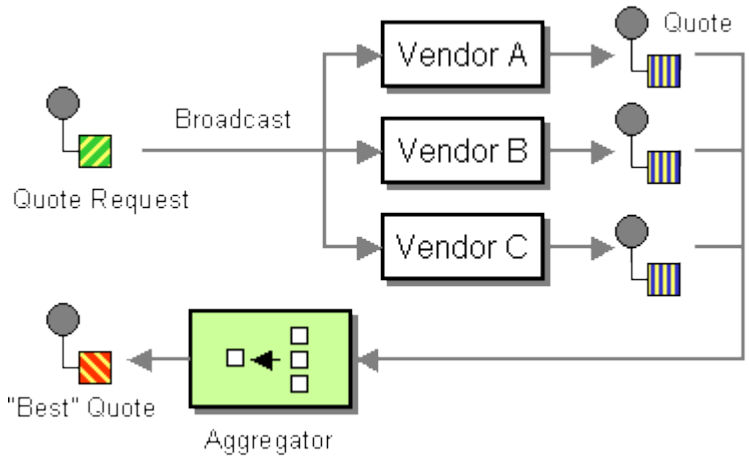
July 2011

Overview

This paper presents a simple and reliable design using the Neuron Pipeline Designer and Runtime to resolve a common scatter-gather application integration problem described in the book Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.

The Scatter Gather pattern is manifested when a requester sends an asynchronous request to a number of providers, which send their replies asynchronously to the requester. A common use for this pattern is when a request for quote is received, and that request must be broadcast out to N number of providers for bid. Each provider will (or may) reply to the request with their respective bid. From there the replies are aggregated and the highest bidder may win.

Another scenario where this pattern is commonly employed is in the context of order processing. In those cases, each order item that is not currently in stock could be supplied by one of multiple external suppliers. However, the suppliers may or may not have the respective item in stock; they may charge a different price and may be able to supply the part by a different date. To fill the order in the best way possible, quotes are requested from all suppliers and then a decision is made as to which one provides the best term for the requested item as show below in Figure 1.



In these scenarios it would not be uncommon if each service that the message was broadcast to required its own unique translation/transformational or other preprocessing requirements before receiving the message. Business rules may need to be injected and, perhaps even the services to broadcast to may need to be dynamically resolved at runtime. In the figure above, each service may represent an individual vendor.
An example of some business requirements addressed by this pattern's implementation may be:

- Contact N airlines simultaneously for price quotes
- Buy ticket from either airline if price<=$$
- Buy the cheapest ticket if price >$$
- Buy the ticket from the first airline to respond

In summary, the Scatter Gather pattern's goal is to send the same message to multiple recipients which will (or may) reply to it. Wait for all (or some) of the replies and aggregate them into a single response message. With this goal in mind, it can be observed that the Scatter Gather pattern includes the Aggregator

(specifically Service Aggregation/Composition) pattern, but may also be composed of several other patterns. For example, when broadcasting the message, a Splitter and Recipient List pattern may be employed.

The Scatter Gather design relies on runtime services, message delivery, correlation, transaction management and web service routing of Neuron ESB.

> **Note:** Service composition, the act of building an application out of multiple services, is usually depicted as a "has a" relationship and the whole is composed of the parts. In contrast, aggregation is a "uses a" type of relationship. The differences are quite subtle but nevertheless important to grasp. In composition relationships, the life cycles of parts are tied to the lifecycle of the whole and when the whole no longer exists, the parts no longer exist either. In aggregation, the parts exist independent of the whole and can go on living after the entity that uses them no longer exists.

This paper (and accompanying sample) establishes how a Neuron Pipeline can be used to broadcast a message, asynchronously, to N number of services (recipients). Within the pipeline, the replies are then aggregated and either published to the bus, or alternatively, they can be returned to the original calling client. The pipeline goes further to illustrate how to dynamically resolve which services to execute at runtime, mapping them to Topics, which are in turn mapped to Neuron Service Connectors. This provides a loosely coupled solution as subscriptions are used to route to the services in the broadcast list. As part of the solution, transformations (XSLT) may be associated with each service, and hence a way to dynamically retrieve a transform from the Neuron ESB Configuration store at runtime and execute it is also demonstrated.

Sample Manifest Description

The accompanying sample is composed of the following 4 components located under the default Neuron installation directory in the following folder: ***\Samples\ServiceTutorials\Scatter Gather Pattern:***

- Neuron ESB Configuration file named **ScatterGatherPipelineSample.esb**. This contains all the Neuron specific artifacts used for this solution (i.e. Topics, Parties, Scatter Gather Pipeline, Client and Service Connectors) pre-configured.
- Visual Studio 2008 solution named **Scatter Gather Solution**. This solution contains the three following Visual Studio C# projects:
    - **ContosoClientRequest** – A console application used to make the client side WCF service request call into the Neuron ESB hosted service endpoint (Neuron Client Connector). The code is displayed in **Code Fragment 2** in the **Designing the Pipeline** section of this document.
    - **NewMartQuoteService** – A WCF service endpoint hosted within a console application. This Web service endpoint represents a Quote Service hosted by a fictional vendor named **New Mart**.
    - **OldMartQuoteService** – A WCF service endpoint hosted within a console application. This Web service endpoint represents a Quote Service hosted by a fictional vendor named **Old Mart**.

At runtime, the sample provides a demonstration of the following scenario, which is dependent upon the implementation of the Scatter Gather pattern described in this document.

Scenario Description

Step 1: *Contoso Supply House* is a fictional, modern, web based company that sells toys, appliances and services to customers over the internet. Contoso hosts their own web site and exposes a public facing web service so external distributors can automate the placement of orders.

Neuron Implementation: The public web service is hosted by Neuron ESB and is configured within the Neuron Client Connector named, *ContosoQuoteService,* exposed on the url, *http://localhost:9001*.

Step 2: An external distributor submits a purchase request for red fire engine toy trucks to Contoso through the public facing web service.

Neuron Implementation: The request is submitted by running the Visual Studio 2008 Console Application named, *ContosoClientRequest.exe*.

Step 3: Contoso receives the distributor's request. However, Contoso doesn't keep any inventory in stock. Instead they submit all orders to external vendors to get the best price for the items requested. The list of vendors to query is selected based on the items ordered or by the distributor's preference.

Neuron Implementation: The request is received by the Neuron Client Connector, ContosoQuoteService. This Client Connector is configured to use the Neuron ESB Publisher named, *ContosoQuoteServicePublisher*, which in turn uses the *Scatter Gather* pipeline described in the *Designing the Pipeline* section of this document.

Step 4: Contoso has an automated process which determines the vendors to query for best price.

Neuron Implementation: The vendors to query are extracted from the custom SOAP header of the original distributor's purchase request from within the pipeline. NOTE: In most real world scenarios, the list of vendors would probably be resolved from a database lookup (or some other means) within the pipeline.

Step 5: Each vendor hosts their own *Quote Service*, a web service that Contoso can submit their distributor's price queries against.

Neuron Implementation: Each vendor's web service is represented by a Neuron Service Connector. There are two vendors setup, *Old Mart* and *New Mart*.
The Old Mart web service endpoint is represented by the following:

- Neuron Service Connector named, "*OldMartQuoteService*".
- OldMartQuoteService receives request messages from the bus through the Neuron ESB Subscriber named, *OldMartQuoteServiceSubscriber*
- OldMartQuoteServiceSubscriber subscribes to the *Finance.Vendors.OldMart.QuoteService* Topic.

The New Mart web service endpoint is represented by the following:

- Service Connector named "*NewMartQuoteService*".
- NewMartQuoteService receives messages from the bus through the Neuron ESB Subscriber named, *NewMartQuoteServiceSubscriber*.
- NewMartQuoteServiceSubscriber subscribes to the *Finance.Vendors.NewMart.QuoteService* Topic.

Note: This describes the inherent mapping that exists between a Topic and Service Endpoint when using Neuron ESB i.e. ***Topic -> Subscriber -> Service Connector -> web service endpoint***.

Step 6: Contoso's automated process submits a query to each vendor's Quote Service, aggregating the results and returning the options to the distributor.

Neuron Implementation: The original distributor's purchase request is transformed according to each vendor's Quote Service requirements. Then the message is published to the bus, asynchronously, to the Topic associated with each vendor's Quote Service.

- For New Mart, the Topic is *Finance.Vendors.NewMart.QuoteService*
- For Old Mart the Topic is *Finance.Vendors.OldMart.QuoteService*

The *Neuron Publishing Service* forwards each request message to the respective web service endpoint of the vendor, waits and then forwards the response from each back to the original instance of the Scatter Gather Pipeline. The responses are combined, optionally transformed and returned to the distributor as a SOAP response message.

Designing the Pipeline

The **Scatter Gather** pipeline depicted in **Figure 2** is built using the following pipeline steps:

- Code, Split, Transform – Xslt, Publish, Decision, Cancel

The pipeline depicted begins with a **Code** pipeline step named "**Get List of Services**".

Get List of Services

The "Get List of Services" responsibility is to determine the list of services to be called, as well as any transformation requirements specific for those services. There are a number of common ways in which this information could be resolved, depending if the information is sent along with the original request, or it is retrieved at runtime. Resolution strategies can include and are not limited to the following:

- Service list information is sent at runtime by client via custom SOAP Headers
- Service list information is maintained in a static list within Neuron
- Service list information is obtained by querying a database at runtime
- Service list information is retrieved through the execution of business rules at runtime

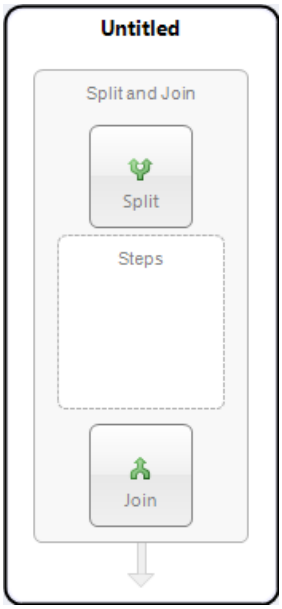


Figure 1: *An empty Split pipeline component as displayed in the Neuron Pipeline designer.*

Figure 2: *The completed Scatter Gather pipeline as displayed in the Neuron Pipeline designer.*

Note: Several pipeline steps (specifically Code, Publish, Split and Decision) within the Neuron Pipeline Designer support direct C# inline editing. In most cases, users simply right click the pipeline step and select "**Edit Code…"** from the short cut menu to display the Code Editing Window (shown in Figure 3). The **Code Editing Window** provides full access to .NET 3x Framework, supports intellisense, displays real time compile errors and supports referencing external .NET assemblies. Compiling pipelines into .NET assemblies is not necessary. All code is dynamically compiled either the first time the component is executed within a pipeline, or after modification.

For example, a list of services to broadcast to, (later aggregate the responses of), could look very similar to the xml fragment below. Each service node has 3 attributes, named topic, action and transform:

```
<NeuronServiceList xmlns='urn:xmlns:neuronesb-com:soapheaders'>
        <services>
                <service topic='Finance.Vendors.OldMart.QuoteService'
                           action=' http://schema.neuron.sample/oldmart/broadcast/IQuote/RequestQuote'
                           transform='QuoteRequest_To_OldMartQuote'>
                </service>
                <service topic='Finance.Vendors.NewMart.QuoteService'
                           action='http://schema.neuron.sample/newmart/broadcast/IBid/RequestBid'
                           transform='QuoteRequest_To_NewMartQuote'>
                </service>
        </services>
</NeuronServiceList>
```

Code Fragment 1: *An example XML document representing a list of services to broadcast an incoming message to. Each service is represented by a "service" node. The topic attribute determines where to route a message is inherently mapped to a service endpoint, represented by a Neuron Service Connector. The action attribute is used to set the Action Header field on the Neuron ESB Message. The transform attribute represents the name of an XSLT stored in the Neuron ESB configuration store.*

In **Code Fragment 1**, a "topic" rather than a "service name" attribute is used to represent the service endpoint that should be called. Neuron ESB uses a Topic based Publish and Subscribe messaging layer to control how messages are routed from service on ramps (Neuron Client Connectors using any WCF binding) to service endpoints (Neuron Service Connectors represent any SOAP based or service URL). This provides a level of abstraction and decoupling which facilitates the ability to easily and quickly change endpoints and routing with minimal impact on clients. The topic attribute is therefore used to represent the actual service endpoint that the client request will be routed to. Within Neuron there is always an underlying mapping created between a service endpoint and Topic when a Subscriber Id (Neuron Party subscribing to one or more Topics) is assigned to a Neuron Service Connector at design time.

The action attribute represents the Action (i.e. Operation/Method) on the service endpoint that needs to be called.

The transform attribute will hold the name of the XSLT stored in the Neuron ESB configuration store. At runtime, the actual XSLT content will be retrieved by using the transform attribute value as the lookup key, passing that key to Neuron's Transform – Xslt Pipeline Step.

Note: The XML format of the service list, property names and namespaces used in this example are intended only for demonstration purposes. In lieu of a static list, a database lookup or LOB (Line of

Business) application query can be used. In fact the most common implementations usually employ some of kind of database lookup or query to retrieve the necessary list of services to call.

For the sake of expediency, this example will use a static list of services defined within the first Code Pipeline Step within the pipeline. WCF (Windows Communication Foundation) can be used to make the initial SOAP request to a generic web service on ramp hosted by Neuron (through configuration of a Neuron Client Connector) as shown in the **Code Fragment 2**:

```
using System;
using System.ServiceModel;
using System.ServiceModel.Channels;
using System.Xml;

namespace Neuron.Esb.Samples
{
    static class Utility
    {
        const string _msg = @"<PurchaseRequest>
                                    <Products>
                                      <Product name='BigBox' quanity='10'
                                            location='Denver'></Product>
                                    </Products>
                              </PurchaseRequest>";


        public static void PipelineByWebService()
        {
            XmlReader xmlReader = null;
            Message reqMsg = null;

            // Sending a request/response to a client connector
            using (var chan = new ChannelFactory<IRequestChannel>(new WSHttpBinding(),
                                        new EndpointAddress("http://localhost:9001")))
            {
                var proxy = chan.CreateChannel();
                using (xmlReader = XmlReader.Create(new System.IO.MemoryStream(
                                        System.Text.Encoding.UTF8.GetBytes(_msg))))
                {
                    using (reqMsg = Message.CreateMessage(
                                        chan.Endpoint.Binding.MessageVersion, "", xmlReader))
                    {
                        Console.WriteLine(proxy.Request(reqMsg).ToString());
                        reqMsg.Close();
                    }
                    xmlReader.Close();
                }
                proxy.Close();
                chan.Close();
            }
        }
    }
}
```

Code Fragment 2: *Sample WCF C# code within Microsoft Visual Studio 2008. This represents code that a client can execute to submit a message to Neuron hosted web service (Neuron Client Connector) listening on port 9001.*

In **Code Fragment 2**, the original request is sent to the Neuron Client Connector's endpoint address (http://localhost:9001). Once the message is received by Neuron, a Code Pipeline Step is used to load the static list of services depicted in **Code Fragment 1** into an Xml Node List object, saving it as a Neuron pipeline variable named, "ServiceConfiguration". By right clicking the Code Pipeline Step named "Get List of Services" within the Scatter Gather pipeline, the C# code fragment can be seen in **Code Fragment 3**:

```
// ***************************************************************************
// First retrieve the list of services to call.  This could come in via soap headers,
// through an external database call, lookup, or from wherever is appropriate for
// the scenario.
// In Neuron, each service is represented by a topic, making this essentially a list of
// topics (or sub topics) to route to, which in turn results in a service call.

// In the last step, we add the list of services as a property to the existing pipeline
// context so that it can be retrieved in the next pipeline step
// ***************************************************************************

string nameSpace = "urn:xmlns:neuronesb-com:soapheaders";
string prefix = "n";
string serviceList =
   @"<NeuronServiceList xmlns='urn:xmlns:neuronesb-com:soapheaders'>
          <services>
            <service topic='Finance.Vendors.OldMart.QuoteService'
              action='http://schema.neuron.sample/oldmart/broadcast/IQuote/RequestQuote'
              transform='QuoteRequest_To_OldMartQuote'></service>
            <service topic='Finance.Vendors.NewMart.QuoteService'
               action='http://schema.neuron.sample/newmart/broadcast/IBid/RequestBid'
               transform='QuoteRequest_To_NewMartQuote'></service>
          </services>
   </NeuronServiceList>";

// Load the list of services into an XML Document, add the namespace and retrieve the
// service nodes,
// persisting the node list into the pipeline's context property
System.Xml.XmlDocument xmlDoc = new System.Xml.XmlDocument();
xmlDoc.LoadXml(serviceList);

System.Xml.XmlNamespaceManager nsMgr = new System.Xml.XmlNamespaceManager(xmlDoc.NameTable);
nsMgr.AddNamespace(prefix, nameSpace);

System.Xml.XmlNodeList nodeList = xmlDoc.SelectNodes("//n:service", nsMgr);
context.Properties.Add("ServiceConfigurations",nodeList);
```

Code Fragment 3: *The Code Editing Window displays the C# code contents of the "Get List of Services" Code pipeline step in the Scatter Gather pipeline example displayed in Figure 2. Code can be edited by right clicking on Code pipeline step and selecting the "Edit Code…" short cut menu.*

In the C# code fragment within the Code step editing window the Xml list of services is loaded into a *System.Xml.XmlDocument* object that an XPATH statement is executed against to retrieve a list of *System.Xml.XmlNodes*. The list of *System.Xml.XmlNodes* represents the list of services submitted by the client. Following this, is the final line of code:

```
context.Properties.Add("ServiceConfigurations",nodeList);
```

Note: The "context" variable represents the *Neuron.Pipelines.PipelineContext* object passed to the Code pipeline step by the Pipeline Runtime. This provides users access to the pipeline context, as well as to the actual ESB Message object (accessed through the Data property).

This last line of code persists the list of services, (and *System.Xml.XmlNodeList* object), as a property of the current pipeline instance (context) so that it may be retrieved later, within other steps in the pipeline. Specifically, the list is saved as a property of the current pipeline context before runtime control is passed to the **Split** pipeline step named "**Broadcast and Aggregate**".

Broadcast and Aggregate

Code Split

The second pipeline step in the Scatter Gather pipeline is the **Code Split** portion of the "Broadcast and Aggregate" Split pipeline step. In this pipeline step the message is cloned into a collection of identical messages, with each message marked to be routed to a specific service contained in the list retrieved from the custom SOAP header.

Here is where it's necessary to understand some of the fundamentals of Neuron Pipelines and the Split pipeline step. Generally, the Split pipeline step allows users to very easily de-batch an incoming message (either by using a simple XPATH expression in a property grid, or by using the Code Editing Window) into their individual parts, submitting these individual parts as a collection of messages to the next stage of the Split pipeline step, called the **Execution Block** (labeled "Steps"). This Execution Block can contain N number of other pipeline steps. An instance of the Execution Block will be executed for each message in the collection of messages submitted to it. Additionally, the Execution Block can be set to run either synchronously (process one message at a time) or asynchronously. In asynchronously mode, the .NET thread pool is utilized to submit each message in the batch to its own Execution Block instance.

The Code Split portion of the Split pipeline step submits a List of *Neuron.Pipelines.PipelineContext* objects, each one containing a message in the batch, to the Execution Block. Since the Split pipeline step "**split type**" property is set to Code, (rather than XPATH), this collection must be created manually by using the following C# code fragment in the Code Split portion of the Split pipeline step:

```
System.Collections.Generic.List<Neuron.Pipelines.PipelineContext<
                                        Neuron.Esb.ESBMessage>> contexts;
contexts = new System.Collections.Generic.List<Neuron.Pipelines.PipelineContext<
                                        Neuron.Esb.ESBMessage>>();
```

After the collection is created, the list of services previously persisted in the "Get List of Services" Code pipeline step must be retrieved as follows:

```
System.Xml.XmlNodeList nodeList =
          (System.Xml.XmlNodeList)context.Properties["ServiceConfigurations"];
```

Once the nodeList object is retrieved, loop through all the nodes performing the following steps:

1. Create a new Neuron ESB Message (*Neuron.Esb.ESBMessage* object), by cloning from the original request message using "*context.Data.Clone()*"
2. Add a custom property to the new ESB Message that contains the Topic to publish to using "*SetProperty()*"
3. Set the Action header of the Neuron ESB Message
4. If transform is required, set the Neuron Transform – Xslt Pipeline Step's dynamic message property using "SetProperty()". This will enable the Transform – Xslt to dynamically look up the Xslt at runtime from Neuron's in memory Xslt configuration store.
5. Create a new *Neuron.Pipelines.PipelineContext* object and add the ESB Message to it
6. Add the new *Neuron.Pipelines.PipelineContext* + ESB Message to the List of *Neuron.Pipelines.PipelineContext* objects

Note: The *Neuron.Esb.Internal.PipelineRuntimeHelper.ClientContext.Configuration* object provides full access to all elements of the Neuron ESB configuration store. This store is loaded into memory by the Neuron ESB Service at startup. This same store is edited by using the Neuron ESB Explorer user interface.

Lastly, the *Neuron.Pipelines.PipelineContext* list is passed to the Execution Block:

```
return contexts;
```

By right clicking the Code pipeline step named "Code Split" within the Split pipeline step, the C# code fragment can be viewed as in **Code Fragment 4**:

```
//Create a list of pipeline contexts to be filled and returned to the aggregator
System.Collections.Generic.List<
        Neuron.Pipelines.PipelineContext<Neuron.Esb.ESBMessage>> contexts;
contexts = new System.Collections.Generic.List<Neuron.Pipelines.PipelineContext<
        Neuron.Esb.ESBMessage>>();

// Retrieve the list of services previously saved to the pipeline context property bag,
System.Xml.XmlNodeList nodeList =
        (System.Xml.XmlNodeList)context.Properties["ServiceConfigurations"];

// Loop through the list of services, we'll create a new message from the original,
// adding the topic to the message's property bag so it can later be retrieved in
// the Steps block. Also add the transform if needed and set the action
foreach(System.Xml.XmlNode node in nodeList)
{

        Neuron.Esb.ESBMessage msg = context.Data.Clone(false);
        msg.SetProperty("neuron","PublishTopic",node.SelectSingleNode("@topic").InnerText);
        msg.SetProperty("neuron","xsltName",node.SelectSingleNode("@transform").InnerText);
        msg.Header.Action = node.SelectSingleNode("@action").InnerText;

        //create new context for the message
        PipelineContext<Neuron.Esb.ESBMessage> splitContext =
                new PipelineContext<Neuron.Esb.ESBMessage> (
                        context.Runtime, context.Pipeline, context.Instance, msg);
        splitContext.Properties.Add("__ClientContext",
                Neuron.Esb.Internal.PipelineRuntimeHelper.ClientContext);

        //add the context to the result
        contexts.Add(splitContext);
}

// Return the batch of messages to be processed by the STEP block
return contexts;
```

Code Fragment 4: *The Code Editing Window displays the C# code contents of the "Code Split" Code pipeline step in the Split pipeline step within the Scatter Gather pipeline example displayed in Figure 2. Code can be edited by right clicking on Code pipeline step and selecting the "Edit Code…" short cut menu.*

Note: *SetProperty()* and *GetProperty()* are methods of the *Neuron.Esb.ESBMessage* object. *SetProperty()* can be used to add or modify existing Neuron specific message properties. It can also be used to add custom or application specific properties (no schema required). A custom property can be added by passing in a prefix, property name and value. *GetProperty()* can later be used to retrieve the value of the property by passing in the prefix and property name.

Steps Execution Block

In the previous step, a new set of cloned messages was created from the original request message, some custom properties were added, a new *Neuron.Pipelines.PipelineContext* was created for each one, and then all were added to a list of *Neuron.Pipelines.PipelineContext* objects which were returned to the pipeline

runtime. Following this, the pipeline runtime will either do one of two things, depending on the value set for the Synchronous property of the Split pipeline step:

1. If the Synchronous property is set to false, then the pipeline runtime will loop through the set of new messages, executing all the pipeline steps within the Execution Block. Each message will be processed, one at a time, completing all the steps in the Execution Block, before the next message can be processed. The processing of all messages is handled on a single thread. For the experienced developer, this is synonymous to a "Foreach" loop construct, whereas all the pipeline steps within the Execution Block are essentially within a Foreach code block.

2. If the Synchronous property is set to true, then the pipeline runtime will use the .NET Thread Pool to pass each message, on its own thread, to an instance of the Execution Block. This is essentially synonymous to the parallel branch execution offered in most workflow designers.

Note: Even though the .NET thread pool is used when the Synchronous property of the Split pipeline step is set to True, there is an internal max limit of 10 threads that will be used.

In the Scatter Gather pipeline, the Synchronous property for the Split pipeline step is set to true.

Once a message is delivered to the Execution Block (labeled "Steps") several things happen, first and foremost a check is made to determine if the original request message must be transformed to a format required by the target service.

Requires Transform

The first pipeline step in the Execution Block is a **Decision** pipeline step, labeled "**Requires Transform?**". This has two branches labeled "**Yes**" and "**No**". The "Yes" branch is configured to return either true or false by evaluating for following inline C# code fragment:

```
return context.Data.GetProperty("neuron","xsltName","").Length > 1;
```

This code fragment retrieves the value of the Transform – Xslt Pipeline Step's "**xsltName**" property (where "**neuron**" is defined as the prefix) which was previously set on the ESB Message (represented by *context.Data*) in the Code Split portion of the Split pipeline step. The Transform – Xslt Pipeline Step optionally will use this property (if set) to dynamically load (by name) the Xslt stored in the Neuron configuration store and execute it. If this line of code evaluates to "true", then all the pipeline steps placed under the "Yes" labeled branch are executed for the ESB Message. If the line of code evaluates to "false", then the "No" labeled branch will be executed. In the Scatter Gather pipeline, the "No" branch does not contain any pipeline steps

Execute Transform

If a transformation is required, control is passed to the Transform – Xslt Pipeline Step labeled "**Dynamic Transform**" located under the "Yes" branch which will translate the original incoming request message (an ESB Message represented by the *context.Data* variable). In our example, XSLT is chosen to translate the message, the name of which was previously retrieved in the Code Split portion of Split pipeline step.

Note: Although the method used in this example to translate the original client request message to the message format required by the service endpoint is XSLT, any method including but not limited to custom code, database query, and LOB query or business rules could be used.

Call Service

Once the transformation logic is completed, the Neuron ESB Message is passed to the **Publish** pipeline step, labeled "**Call Service**". The underlying purpose of this pipeline step is to call the service endpoint, represented by a Neuron Service Connector. As was pointed out earlier in this document, our solution is dependent on associating each service endpoint with (i.e. subscribing to) a specific Topic (or Sub Topic).

Therefore the Publish pipeline step publishes the Neuron ESB Message to the Topic by using the value of the custom "**PublishTopic**" property (where "**NEURON**" is defined as the prefix) which was previously set on the ESB Message (represented by *context.Data*) in the Code Split portion of the Split pipeline step. In the Scatter Gather pipeline sample, the Publish pipeline step's **SelectorType** property is configured to resolve the Topic to publish to by using the Code editor as follows:

```
// Set the topic of the Publish step by retrieving it from custom message property
return message.GetProperty("neuron","PublishTopic","");
```

Since a response message is expected from the service subscribing to the Topic published on, the Publish pipeline step's **Semantic** property is set to **Request**. This will cause the pipeline execution to wait for either the response message (forwarded from the Neuron Service Connector by the Neuron Publishing Service) or a service timeout. Once the response message is received by the executing pipeline instance, the pipeline runtime will use the contents of the response message to replace the body of the current ESB Message being processed. This newly modified ESB Message will be used in all subsequent pipeline steps.

Behind the scenes, the **Neuron Publishing Service** accepts the published message, forwards it to the expecting service endpoint, calls the endpoint, retrieves the response message and forwards that response (through internal correlation) back to the original instance of the running pipeline. One of the many benefits of having the Neuron Publishing Service (commonly referred to as the Neuron ESB Bus) sitting between the pipeline and the service endpoint is that additional subscribers can be added at any time to the Neuron configuration without affecting either existing clients or the service endpoints. Additionally, the details of the service endpoint can be changed as well (i.e. WCF binding, URL, etc.) without impact on the clients submitting messages to the system. If the Schema for the service endpoint changed, then only the transform logic or XSLT would have to be updated, nothing would have to be recompiled or decomposed.

Join

Once ALL the messages are processed by their respective instance of the Execution Block and its associated pipeline steps, control is passed to the **Join** portion of the "Broadcast and Aggregate" Split pipeline step. Within the Join, the pipeline runtime can either reassemble all the individual Neuron ESB messages which were processed by the Execution Block back into a list of *Neuron.Pipelines.PipelineContext* objects, or discard them. How the Join functions is determined by the value of the "**join type**" property. In the Scatter Gather pipeline, the join type is set to **Wrapper**, which allows configuration of the Join by setting two

properties in property grid, **WrapperElementName** and **WrapperElementNamespace.** In our example the following property values were used:

```
WrapperElementName = QuoteCollection
WrapperElementNamespace = http://schema.neuron.sample/broadcast/result
```

This results in all the Neuron ESB Message bodies contained in the list of *Neuron.Pipelines.PipelineContext* objects to be combined and encapsulated by an XML root node of "**QuoteCollection**" with the default namespace of "**http://schema.neuron.sample/broadcast/result".** An example of a result that the Join would produce from the original message request follows:

```
<QuoteCollection xmlns="http://schema.neuron.sample/broadcast/result">
      <QuoteResult>Response received from service endpoint 1…</QuoteResult>
      <QuoteResult>Response received from service endpoint 2…</QuoteResult>
</QuoteCollection>
```

Code Fragment 5: *An XML document representing the aggregated result message returned from the list of services that were executed. Each service response is represented by a <QuoteResult> node. However, a service can return ANY XML data. Whatever data is returned is inserted as a child node under the QuoteCollection wrapper root element.*

In **Code Fragment 5**, each <QuoteResult> node represents the returned response message from a service called through the "Call Service" Publish pipeline step. In our example, if further modification to the aggregated result message is needed, the Split pipeline step could be followed by another pipeline step, perhaps either a **Transform – Xslt**, **Code** or **Rules-WF** pipeline step.

However, more granular control over the collection of Neuron ESB Messages returned from the Execution Block can be obtained by changing the join type property from Wrapper to **Code**. This results in access to the Code Editor for direct modification of the collection. An example of C# code used within a Join is shown in **Code Fragment 6**:

```
using (System.IO.StringWriter sw = new System.IO.StringWriter())
{
    using (XmlTextWriter xw = new XmlTextWriter(sw))
    {
       //writes <QuoteCollection> root element
       xw.WriteStartElement("QuoteCollection");

        foreach (PipelineContext<Neuron.Esb.ESBMessage> c in splits)
        {
            //adds <QuoteResult>...</QuoteResult>
            xw.WriteRaw(c.Data.ToXml());
        }

        //writes </QuoteCollection>
        xw.WriteEndElement();
        xw.Flush();
    }
    //Replace original request message with new aggregated response message.
    Neuron.Esb.ESBMessage esbMsg = new Neuron.Esb.ESBMessage();
    esbMsg.FromXml(sw.ToString());
    context.Data = esbMsg;
}
```

Code Fragment 6: *The code sample demonstrates how to aggregate the responses returned from all the service calls using an XmlTextWriter. Each service call response is represented by a PipelineContext object (c) within the splits collection. This is exactly what happens when using the Wrapper join type.*

When using the Code join type, the original *Neuron.Pipelines.PipelineContext* (which contains the original unaltered client response *Neuron.Esb.ESBMessage*) as well the list of *Neuron.Pipelines.PipelineContext* objects processed by the Execution Block, are passed into the Code Editor as arguments represented by the variables "**context**" and "**splits"**. As shown in Code Fragment 5 above, entirely custom xml can be created to aggregate the response messages collected from the Execution Block by iterating through the splits collection using a foreach construct.

Lastly, to return the newly created aggregated XML response message to the remaining steps of the pipeline, the Neuron.Esb.ESBMessage assigned to the current context (represented by the context.Data variable) must be replaced with a new Neuron.Esb.ESBMessage containing the aggregated XML.

Return Result

The final piece of the pipeline is a **Cancel** pipeline step named "***Return Result***". This is where a decision will be made regarding what to do with the aggregated response message returned from the Join. In the Scatter Gather pipeline example, a Cancel pipeline step is used because the response message is being returned to the calling client. Using a Cancel pipeline step essentially stops the response message from continuing beyond the pipeline instance and being published to the bus. Instead, the response message is returned to the original calling client. This will happen AS LONG AS THE ORIGINAL REQUEST IS A REQUEST/RESPONSE TYPE OF REQUEST. This is configured by either manually setting the **Semantic** property of the original Neuron.Esb.Message to "**Request**" or automatically, by ensuring that the Neuron Client Connector's **Messaging Pattern** property located on the **Binding** tab is set to "**Request-Reply**" rather than "**Datagram**".

However if required, the response message can be forwarded to the original Topic, or another Topic by either removing this step, or replacing it with a Publish pipeline step. In this latter case, the Neuron Client Connector's Messaging Pattern should be set to Datagram, and the associated code to send the client request to Neuron (shown in Code Fragment 2) should be altered appropriately using the **IOutputChannel** (or similar interface) so as to avoid client timeout errors.

Configuring the Solution

The accompanying Neuron ESB Configuration file named, **ScatterGatherPipelineSample.esb**, is configured to support the Scatter Gather solution described in this paper. The configuration file can be opened within the Neuron ESB Explorer. Within it are the following elements:

Neuron Topics

A specific Topic topology was created to support publishing the original request message to the bus, as well as routing the message to multiple web services (service endpoints represented by Neuron Service Connectors), each hosted by a different company vendor, specifically "**New Mart**" and "**Old Mart**". These company vendors are responsible for responding to the incoming quote request submitted by the client. One web service is associated with the **Finance.Vendors.NewMart.QuoteService** (the "New Mart" vendor) Topic, while the other is associated with the **Finance.Vendors.OldMart.QuoteService** (the "Old Mart" vendor) Topic. When using Topics to route to service endpoints it's important to use an understandable Topic Topology. This will allow for more intuitive mappings between service endpoints and Topics i.e. ***Topic -> Subscriber -> Service Connector -> web service endpoint***

The following Topics are configured under the ***Messaging:Topics:Topics*** section of the Neuron ESB Explorer:

```
Root Topic: Finance
Sub Topics: Finance.Purchases
            Finance.Vendors
            Finance.Vendors.NewMart
            Finance.Vendors.NewMart.QuoteService
            Finance.Vendors.OldMart
            Finance.Vendors.OldMart.QuoteService
```

All distributor purchase requests are originally submitted to the **Finance.Purchases** Topic, while all price quote requests to Vendors are submitted to **Finance.Vendors.<*vendorName*>.QuoteService**.

Neuron Parties

Neuron ESB Parties are used to communicate to, and receive messages from the Neuron ESB Publishing Services. Every Topic represents an instance of a Neuron ESB Publishing Service. What messages a Neuron ESB Party is allowed to send, and what messages they can receive is determined by what subscriptions are assigned to them. Subscriptions are generally defined by using Topics with the optional addition of **Message Patterns.**

> **Note:** Message Patterns provide more granularity to Topic based subscriptions through the addition of context and content based routing options.

Neuron ESB Parties can be defined as either a **Publisher**, **Subscriber** or both. How they are defined is a function of the Topic subscription rights assigned to them, **Send**, **Receive** or both.

In the scenario outlined in this paper, the distributor submits a purchase request to the service endpoint hosted by Contoso. This service endpoint is represented by a Neuron Client Connector, **ContosoQuoteService**. The Neuron Client Connector is configured to submit all incoming purchase request

messages to the bus using a Neuron ESB Publisher. That in turn is configured to run the Scatter Gather pipeline before messages are published to the bus. To support this, the following **Publisher** is configured under the ***Messaging:Topics:Publishers*** section of the Neuron ESB Explorer and assigned to the Neuron Client Connector:

```
Name:            ContosoQuoteServicePublisher
Subscriptions:   Finance.Purchases                  Send
                 Finance.Vendors.*.QuoteService     Send
Pipelines:       Scatter Gather                     On Publish event
```

> **Note:** When using Topics and sub Topics, wildcard subscription notation (*) can be used. In the example above, Finance.Vendors.*.QuoteService would allow the publisher to submit and receive messages from ANY vendor specified between the two periods (.). The subscription is resolved dynamically at runtime.

Each vendor, Old Mart and New Mart, maintain their own Quote Service, represented by a Neuron Service Connector. To receive a purchase request message from the Scatter Gather pipeline, each one is configured to use a specific **Neuron ESB Subscriber**. The respective Subscriber receives messages from the Topic its configured for, and forwards those messages to the Neuron Service Connector, which in turn, calls the vendor's Quote Service endpoint url. To support this, the following **Subscribers** are configured under the ***Messaging:Topics:Subscribers*** section of the Neuron ESB Explorer, each assigned to their respective Neuron Service Connector:

```
Name:            NewMartQuoteServiceSubscriber
Subscriptions:   Finance.Vendors.NewMart.QuoteService     Receive

Name:            OldMartQuoteServiceSubscriber
Subscriptions:   Finance.Vendors.OldMart.QuoteService     Receive
```

Neuron Service Endpoints

Neuron Service Endpoints are used to either host service endpoints, (essentially using standardized web service protocols to expose a Topic publishing service to receive messages) or, to communicate to existing service endpoint urls, (Neuron sends a message from the bus to the specific service endpoint url). The former is regarded as a **Neuron Client Connectors**, whereas the latter is a **Neuron Service Connector**. Both are a type of Neuron Service Endpoint.

In the scenario outlined in this paper, the distributor submits a purchase request to the service endpoint hosted by Contoso.

To support this, the following **Neuron Client Connector** (representing the Contoso endpoint) is configured under the ***Connections:Endpoints:Service Endpoints*** section of the Neuron ESB Explorer:

```
General tab: Name:                  ContosoQuoteService
Client Connector tab: URL:          http://localhost:9001
Client Connector tab: Publisher Id: ContosoQuoteServicePublisher
Client Connector tab: Topic:        Finance.Purchases
General tab: Binding:               WSHttp
Security tab: Security:             Message:Windows
```

```
Client Connector tab: Capture custom headers
```

The following **Neuron Service Connectors**, representing the existing Quote Services for each respective vendor, are configured under the ***Connections:Endpoints:Service Endpoints*** section of the Neuron ESB Explorer:

```
Name:             NewMartQuoteService
URL:              http://localhost:8732/
Subscriber Id:    NewMartQuoteServiceSubscriber
Binding:          BasicHttp

Name:             OldMartQuoteService
URL:              http://localhost:8731/
Subscriber Id:    OldMartQuoteServiceSubscriber
Binding:          BasicHttp
```

Neuron Data Store

XML documents, XSD Schemas and XSL Transformation documents can be persisted and referenced from the Neuron Data Store configured under the ***Data*** section of the Neuron ESB Explorer. XSD Schemas and XSL Transformation documents can be referenced directly within the Validate – Schema and Transform - XSLT pipeline steps. Additionally, the Neuron Data Store can be accessed at runtime through the *Neuron.Esb.Internal.PipelineRuntimeHelper.ClientContext.Configuration* object to dynamically retrieve stored documents.

In this sample, the following XML Documents at stored only for referenced. They are not retrieved at runtime.

```
Name:             ClientPurchaseRequest
Description:      Sample purchase request submitted by distributors
                  to the Contoso web service

Name:             NewMartQuoteRequest
Description:      Sample Quote request message expected by the New
                  Mart Quote Service

Name:             OldMartQuoteRequest
Description:      Sample Quote request message expected by the Old
                  Mart Quote Service

Name:             QuoteResult
Description:      Sample Quote result message returned by both Old
                  Mart and New Mart Quote Services
```

The following XSL Transformation documents are retrieved at runtime to transform the incoming purchase request to the expected format of the vendor's Quote Service.

```
Name:       QuoteRequest_To_NewMartQuote
```

Name: QuoteRequest_To_OldMartQuote

Running the Solution

Open the Neuron ESB Configuration

1. Unzip the **NeuronSamples.zip** located in /Samples directory under the default Neuron installation directory. Navigate to the following folder: ***\Samples\ServiceTutorials\Scatter Gather Pattern***
2. Start **Neuron ESB Explorer**; select the ***Open*** radio button on the ***Open ESB Configuration*** dialog and specify the **ScatterGatherPipelineSample.esb** configuration file in the ***File Location*** panel.
3. Set the Neuron ESB Service to use this configuration by selecting the ***Configure Server*** menu item and specifying the **ScatterGatherPipelineSample.esb** configuration file. Restart the Neuron ESB Service using the ***Server Status*** dropdown menu item.
4. Close the current ESB Configuration file within Neuron ESB Explorer and immediately select the ***Connect*** radio button.

Setup the Quote Services

1. Open the Visual Studio 2008 **Scatter Gather Solution.sln** solution and rebuild it.
2. Navigate to the **NewMartQuoteService** Visual Studio 2008 project's BIN\DEBUG directory and run the **NewMartQuoteService.exe** Console Application. This hosts the New Mart Quote Service.
3. Navigate to the **OldMartQuoteService** Visual Studio 2008 project's BIN\DEBUG directory and run the **OldMartQuoteService.exe** Console Application. This hosts the Old Mart Quote Service.

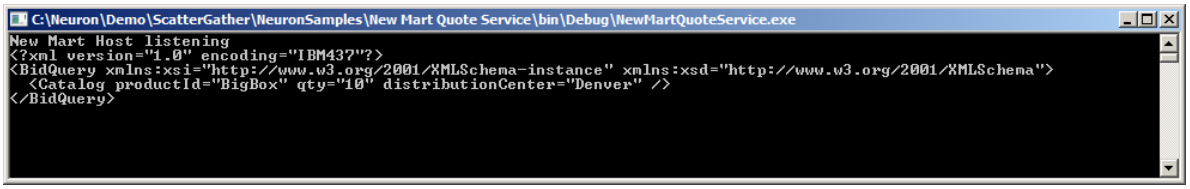
Run the Sample

1. Navigate to the **ContosoClientRequest** Visual Studio 2008 project's BIN\DEBUG directory and run the **ContosoClientRequest.exe** Console Application. This will submit the purchase request to the Neuron Client Connector. Once the console application is running you **MUST** press the **ENTER** key on the keyboard to submit the request to Neuron.

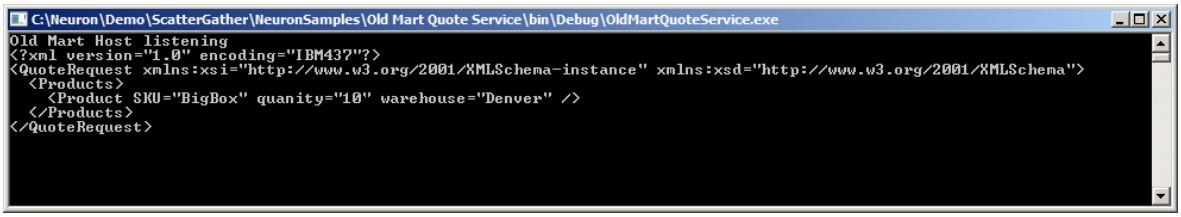
Results

Once a distributor submits a purchase request (by running the ContosoClientRequest.exe Console Application and pressing ENTER) to the Neuron hosted service endpoint (Neuron Client Connector), the Scatter Gather pipeline is executed. The pipeline resolves the vendor service endpoints, transforms the original purchase request as necessary, sending it to individual Quote Services of New Mart and Old Mart

1. A transformed purchase request will be written to the Console Application window hosting the NewMartQuoteService as in the figure below:

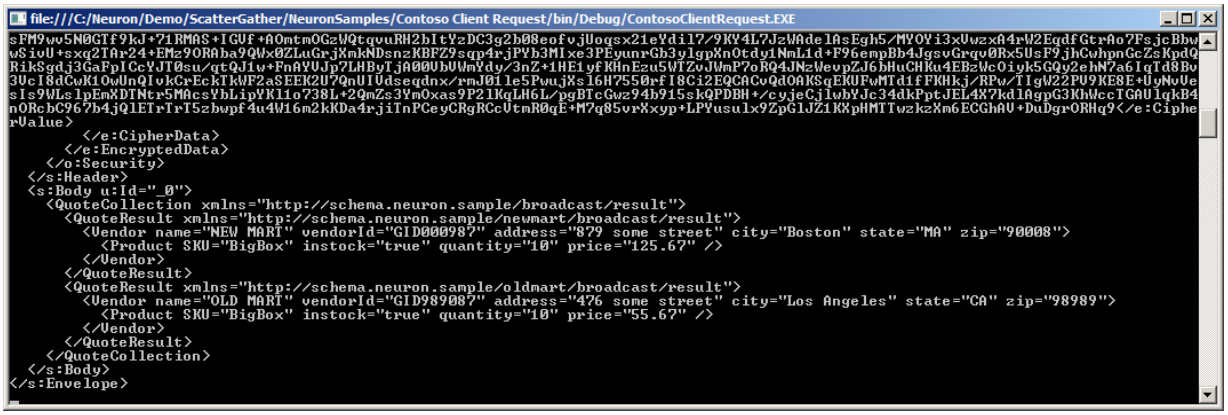


2. A transformed purchase request will be written to the Console Application window hosting the OldMartQuoteService as in the figure below:



Each vendor's Quote Service will return a result. These results are aggregated within the pipeline and returned to the distributor making the purchase request. This is represented by the result written to the Console Application window as in the figure below:

1. The returned results from both Quote Services are enclosed within the **QuoteCollection** element and written to the Console Application window that made the original purchase request. This also includes the full WSHttp envelope.



Pipeline Designer

All pipelines with the exception of the Code pipeline step are configured by selecting and setting their properties in the property grid located at the bottom right of the pipeline designer. The Code pipeline step is configured by selecting the “Edit Code…” option from the short cut menu that is available when right-clicking the Code step in the pipeline designer. See the pipeline documentation for more information.

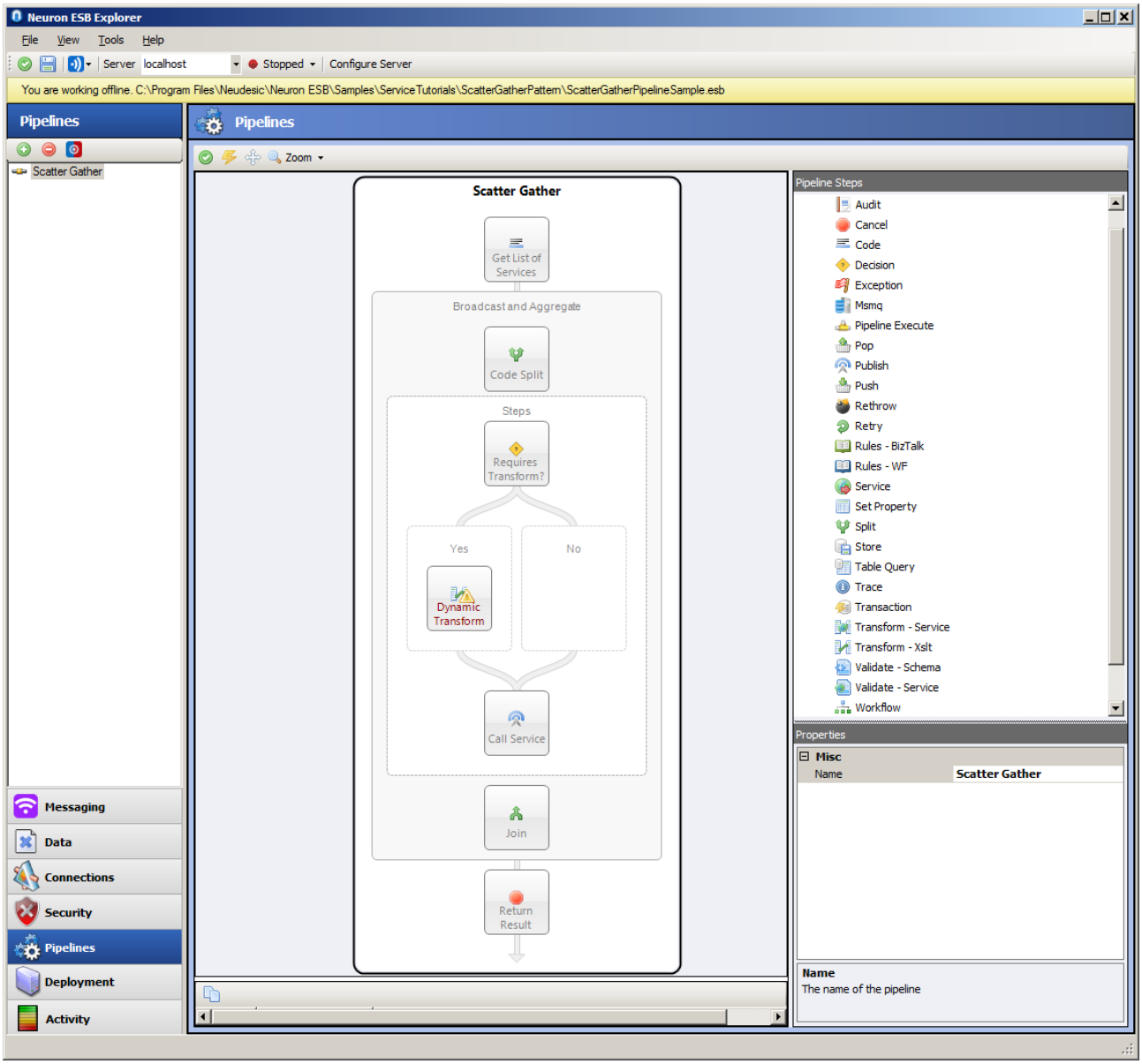


Figure 3: *The Neuron Pipeline Designer displaying the Scatter Gather pipeline. Property Grid at the bottom right displaying the properties of Split pipeline step named “Broadcast and Aggregate”. Note the Synchronous property is set to False, allowing a broadcast into the Execution Block (labeled “Steps”) for processing rather than each message processed one at time”.*