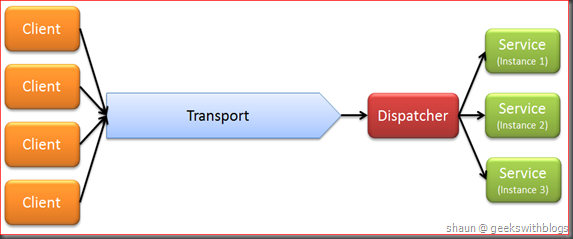
## [Scaling-out Your Services by Message Bus based WCF Transport Extension – Part 1 – Background](http://geekswithblogs.net/shaunxu/archive/2012/03/23/scaling-out-your-services-by-message-bus-based-wcf-transport-extension.aspx" \o "Title of this entry.)

Cloud computing gives us more flexibility on the computing resource, we can provision and deploy an application or service with multiple instances over multiple machines. With the increment of the service instances, how to balance the incoming message and workload would become a new challenge.

Currently there are two approaches we can use to pass the incoming messages to the service instances, I would like call them dispatcher mode and pulling mode.

## Dispatcher Mode

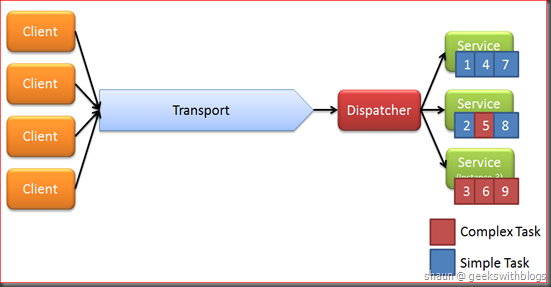
The dispatcher mode introduces a role which takes the responsible to find the best service instance to process the request. The image below describes the sharp of this mode.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_4.png)

There are four clients communicate with the service through the underlying transportation. For example, if we are using HTTP the clients might be connecting to the same service URL. On the server side there’s a dispatcher listening on this URL and try to retrieve all messages. When a message came in, the dispatcher will find a proper service instance to process it. There are three mechanism to find the instance:

* Round-robin: Dispatcher will always send the message to the next instance. For example, if the dispatcher sent the message to instance 2, then the next message will be sent to instance 3, regardless if instance 3 is busy or not at that moment.
* Random: Dispatcher will find a service instance randomly, and same as the round-robin mode it regardless if the instance is busy or not.
* Sticky: Dispatcher will send all related messages to the same service instance. This approach always being used if the service methods are state-ful or session-ful.

But as you can see, all of these approaches are not really load balanced. The clients will send messages at any time, and each message might take different process duration on the server side. This means in some cases, some of the service instances are very busy while others are almost idle. For example, if we were using round-robin mode, it could be happened that most of the simple task messages were passed to instance 1 while the complex ones were sent to instance 3, even though instance 1 should be idle.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_6.png)

This brings some problem in our architecture. The first one is that, the response to the clients might be longer than it should be. As it’s shown in the figure above, message 6 and 9 can be processed by instance 1 or instance 2, but in reality they were dispatched to the busy instance 3 since the dispatcher and round-robin mode.

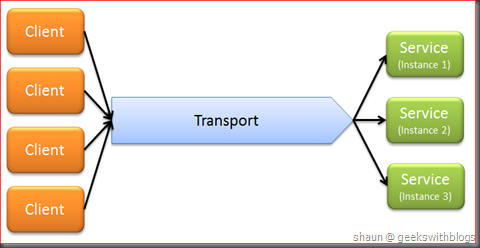
Secondly, if there are many requests came from the clients in a very short period, service instances might be filled by tons of pending tasks and some instances might be crashed.

Third, if we are using some cloud platform to host our service instances, for example the Windows Azure, the computing resource is billed by service deployment period instead of the actual CPU usage. This means if any service instance is idle it is wasting our money!

Last one, the dispatcher would be the bottleneck of our system since all incoming messages must be routed by the dispatcher. If we are using HTTP or TCP as the transport, the dispatcher would be a network load balance. If we wants more capacity, we have to scale-up, or buy a hardware load balance which is very expensive, as well as scaling-out the service instances.

## Pulling Mode

Pulling mode doesn’t need a dispatcher to route the messages. All service instances are listening to the same transport and try to retrieve the next proper message to process if they are idle.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_8.png)

Since there is no dispatcher in pulling mode, it requires some features on the transportation.

* The transportation must support multiple client connection and server listening. HTTP and TCP doesn’t allow multiple clients are listening on the same address and port, so it cannot be used in pulling mode directly.
* All messages in the transportation must be FIFO, which means the old message must be received before the new one.
* Message selection would be a plus on the transportation. This means both service and client can specify some selection criteria and just receive some specified kinds of messages. This feature is not mandatory but would be very useful when implementing the request reply and duplex WCF channel modes. Otherwise we must have a memory dictionary to store the reply messages. I will explain more about this in the following articles.

Message bus, or the message queue would be best candidate as the transportation when using the pulling mode. First, it allows multiple application to listen on the same queue, and it’s FIFO. Some of the message bus also support the message selection, such as [TIBCO EMS](http://www.tibco.com/products/soa/messaging/enterprise-mesage-service/default.jsp" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank), [RabbitMQ](http://www.rabbitmq.com/" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank). Some others provide in memory dictionary which can store the reply messages, for example the [Redis](http://redis.io/" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank).

The principle of pulling mode is to let the service instances self-managed. This means each instance will try to retrieve the next pending incoming message if they finished the current task. This gives us more benefit and can solve the problems we met with in the dispatcher mode.

* The incoming message will be received to the best instance to process, which means this will be very balanced. And it will not happen that some instances are busy while other are idle, since the idle one will retrieve more tasks to make them busy.
* Since all instances are try their best to be busy we can use less instances than dispatcher mode, which more cost effective.
* Since there’s no dispatcher in the system, there is no bottleneck.
* When we introduced more service instances, in dispatcher mode we have to change something to let the dispatcher know the new instances. But in pulling mode since all service instance are self-managed, there no extra change at all.
* If there are many incoming messages, since the message bus can queue them in the transportation, service instances would not be crashed.

All above are the benefits using the pulling mode, but it will introduce some problem as well.

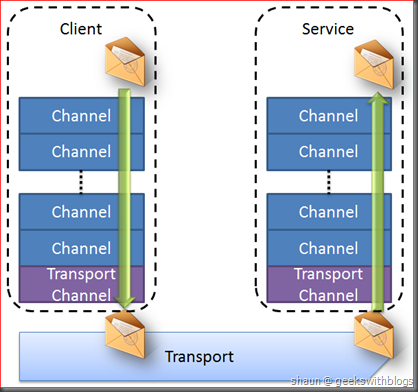
* The process tracking and debugging become more difficult. Since the service instances are self-managed, we cannot know which instance will process the message. So we need more information to support debug and track.
* Real-time response may not be supported. All service instances will process the next message after the current one has done, if we have some real-time request this may not be a good solution.

Compare with the Pros and Cons above, the pulling mode would a better solution for the distributed system architecture. Because what we need more is the scalability, cost-effect and the self-management.

## WCF and WCF Transport Extensibility

Windows Communication Foundation (WCF) is a framework for building service-oriented applications. In the .NET world WCF is the best way to implement the service. In this series I’m going to demonstrate how to implement the pulling mode on top of a message bus by extending the WCF.

I don’t want to deep into every related field in WCF but will highlight its transport extensibility. When we implemented an RPC foundation there are many aspects we need to deal with, for example the message encoding, encryption, authentication and message sending and receiving. In WCF, each aspect is represented by a channel. A message will be passed through all necessary channels and finally send to the underlying transportation. And on the other side the message will be received from the transport and though the same channels until the business logic.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_12.png)

This mode is called “Channel Stack” in WCF, and the last channel in the channel stack must always be a transport channel, which takes the responsible for sending and receiving the messages. As we are going to implement the WCF over message bus and implement the pulling mode scaling-out solution, we need to create our own transport channel so that the client and service can exchange messages over our bus. Before we deep into the transport channel, let’s have a look on the message exchange patterns that WCF defines.

Message exchange pattern (MEP) defines how client and service exchange the messages over the transportation. WCF defines 3 basic MEPs which are datagram, Request-Reply and Duplex.

* Datagram: Also known as one-way, or fire-forgot mode. The message sent from the client to the service, and no need any reply from the service. The client doesn’t care about the message result at all.
* Request-Reply: Very common used pattern. The client send the request message to the service and wait until the reply message comes from the service.
* Duplex: The client sent message to the service, when the service processing the message it can callback to the client. When callback the service would be like a client while the client would be like a service.

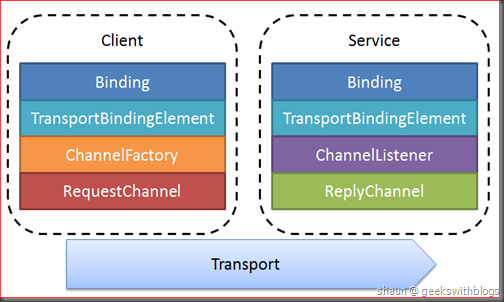
In WCF, each MEP represent some channels associated.

|  |  |
| --- | --- |
| **MEP** | **Channels** |
| Datagram | IInputChannel, IOutputChannel |
| Request-Reply | IRequestChannel, IReplyChannel |
| Duplex | IDuplexChannel |

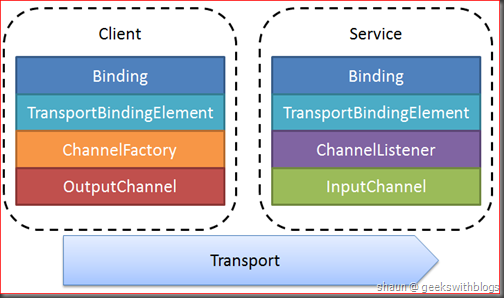
**And the channels are created by ChannelListener on the server side, and ChannelFactory on the client side**. **The ChannelListener and ChannelFactory are created by the TransportBindingElement.** The **TransportBindingElement is created by the Binding**, which can be defined as a new binding or from a custom binding.

For more information about the transport channel mode, please refer to the [MSDN document](http://msdn.microsoft.com/en-us/library/ms729840.aspx" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank).

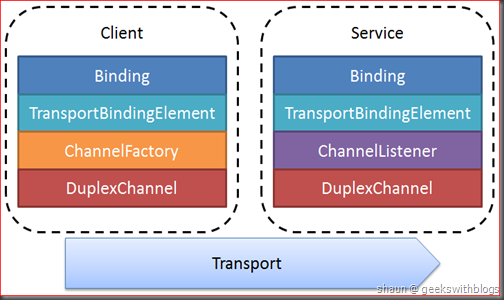
The figure below shows the transport channel objects when using the request-reply MEP.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_14.png)

And this is the datagram MEP.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_16.png)

And this is the duplex MEP.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scale-out-WCF-Services-via-MQ-by-Transpo_C5DF/image_18.png)

After investigated the WCF transport architecture, channel mode and MEP, we finally identified what we should do to extend our message bus based transport layer. They are:

* Binding: (Optional) Defines the channel elements in the channel stack and added our transport binding element at the bottom of the stack. But we can use the build-in CustomBinding as well.
* TransportBindingElement: Defines which MEP is supported in our transport and create the related ChannelListener and ChannelFactory. This also defines the scheme of the endpoint if using this transport.
* ChannelListener: Create the server side channel based on the MEP it’s. We can have one ChannelListener to create channels for all supported MEPs, or we can have ChannelListener for each MEP. In this series I will use the second approach.
* ChannelFactory: Create the client side channel based on the MEP it’s. We can have one ChannelFactory to create channels for all supported MEPs, or we can have ChannelFactory for each MEP. In this series I will use the second approach.
* Channels: Based on the MEPs we want to support, we need to implement the channels accordingly. For example, if we want our transport support Request-Reply mode we should implement IRequestChannel and IReplyChannel. In this series I will implement all 3 MEPs listed above one by one.
* Scaffold: In order to make our transport extension works we also need to implement some scaffold stuff. For example we need some classes to send and receive message though out message bus. We also need some codes to read and write the WCF message, etc.. These are not necessary but would be very useful in our example.

## Message Bus

There is only one thing remained before we can begin to implement our scaling-out support WCF transport, which is the message bus. As I mentioned above, the message bus must have some features to fulfill all the WCF MEPs. In my company we will be using TIBCO EMS, which is an enterprise message bus product. And I have said before we can use any message bus production if it’s satisfied with our requests.

Here I would like to introduce an interface to separate the message bus from the WCF. This allows us to implement the bus operations by any kinds bus we are going to use. The interface would be like this.

1: public interface IBus : IDisposable

2: {

3: string SendRequest(string message, bool fromClient, string from, string to = null);

4:

5: void SendReply(string message, bool fromClient, string replyTo);

6:

7: BusMessage Receive(bool fromClient, string replyTo);

8: }

There are only three methods for the bus interface. Let me explain one by one.

The SendRequest method takes the responsible for sending the request message into the bus. The parameters description are:

* message: The WCF message content.
* fromClient: Indicates if this message was came from the client.
* from: The channel ID that this message was sent from. The channel ID will be generated when any kinds of channel was created, which will be explained in the following articles.
* to: The channel ID that this message should be received. In Request-Reply and Duplex MEP this is necessary since the reply message must be received by the channel which sent the related request message.

The SendReply method takes the responsible for sending the reply message. It’s very similar as the previous one but no “from” parameter. This is because it’s no need to reply a reply message again in any MEPs.

The Receive method takes the responsible for waiting for a incoming message, includes the request message and specified reply message. It returned a BusMessage object, which contains some information about the channel information. The code of the BusMessage class is

1: public class BusMessage

2: {

3: public string MessageID { get; private set; }

4: public string From { get; private set; }

5: public string ReplyTo { get; private set; }

6: public string Content { get; private set; }

7:

8: public BusMessage(string messageId, string fromChannelId, string replyToChannelId, string content)

9: {

10: MessageID = messageId;

11: From = fromChannelId;

12: ReplyTo = replyToChannelId;

13: Content = content;

14: }

15: }

Now let’s implement a message bus based on the IBus interface. Since I don’t want you to buy and install the TIBCO EMS or any other message bus products, I will implement an in process memory bus. This bus is only for test and sample purpose. It can only be used if the service and client are in the same process. Very straightforward.

1: public class InProcMessageBus : IBus

2: {

3: private readonly ConcurrentDictionary<Guid, InProcMessageEntity> \_queue;

4: private readonly object \_lock;

5:

6: public InProcMessageBus()

7: {

8: \_queue = new ConcurrentDictionary<Guid, InProcMessageEntity>();

9: \_lock = new object();

10: }

11:

12: public string SendRequest(string message, bool fromClient, string from, string to = null)

13: {

14: var entity = new InProcMessageEntity(message, fromClient, from, to);

15: \_queue.TryAdd(entity.ID, entity);

16: return entity.ID.ToString();

17: }

18:

19: public void SendReply(string message, bool fromClient, string replyTo)

20: {

21: var entity = new InProcMessageEntity(message, fromClient, null, replyTo);

22: \_queue.TryAdd(entity.ID, entity);

23: }

24:

25: public BusMessage Receive(bool fromClient, string replyTo)

26: {

27: InProcMessageEntity e = null;

28: while (true)

29: {

30: lock (\_lock)

31: {

32: var entity = \_queue

33: .Where(kvp => kvp.Value.FromClient == fromClient && (kvp.Value.To == replyTo || string.IsNullOrWhiteSpace(kvp.Value.To)))

34: .FirstOrDefault();

35: if (entity.Key != Guid.Empty && entity.Value != null)

36: {

37: \_queue.TryRemove(entity.Key, out e);

38: }

39: }

40: if (e == null)

41: {

42: Thread.Sleep(100);

43: }

44: else

45: {

46: return new BusMessage(e.ID.ToString(), e.From, e.To, e.Content);

47: }

48: }

49: }

50:

51: public void Dispose()

52: {

53: }

54: }

The InProcMessageBus stores the messages in the objects of InProcMessageEntity, which can take some extra information beside the WCF message itself.

1: public class InProcMessageEntity

2: {

3: public Guid ID { get; set; }

4: public string Content { get; set; }

5: public bool FromClient { get; set; }

6: public string From { get; set; }

7: public string To { get; set; }

8:

9: public InProcMessageEntity()

10: : this(string.Empty, false, string.Empty, string.Empty)

11: {

12: }

13:

14: public InProcMessageEntity(string content, bool fromClient, string from, string to)

15: {

16: ID = Guid.NewGuid();

17: Content = content;

18: FromClient = fromClient;

19: From = from;

20: To = to;

21: }

22: }

**Summary**

OK, now I have all necessary stuff ready. The next step would be implementing our WCF message bus transport extension.

In this post I described two scaling-out approaches on the service side especially if we are using the cloud platform: **dispatcher mode** and **pulling mode**. And I compared the Pros and Cons of them. Then I introduced the WCF channel stack, channel mode and the transport extension part, and identified what we should do to create our own WCF transport extension, to let our WCF services using pulling mode based on a message bus. And finally I provided some classes that need to be used in the future posts that working against an in process memory message bus, for the demonstration purpose only.

In the next post I will begin to implement the transport extension step by step.

Hope this helps,

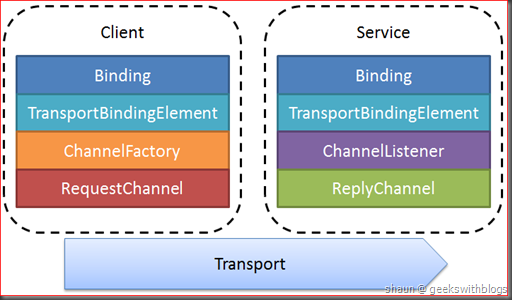
Shaun

<http://geekswithblogs.net/shaunxu/archive/2012/03/23/scaling-out-your-services-by-message-bus-based-wcf-transport-extension.aspx>

## [Scaling-out Your Services by Message Bus based WCF Transport Extension – Part 2 – Frist Trying Request-Reply Mode](http://geekswithblogs.net/shaunxu/archive/2012/03/23/scaling-out-your-services-by-message-bus-based-wcf-transport-extension-again.aspx" \o "Title of this entry.)

In my [previous post](http://geekswithblogs.net/shaunxu/archive/2012/03/23/scaling-out-your-services-by-message-bus-based-wcf-transport-extension.aspx" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank) I introduced the architecture of message bus based system, the dispatcher mode and the pulling mode. I also explained a bit about the channel mode and transport extensibility of WCF. And then, in order to make the following sample simple and easy to use I created an in process and in memory message bus.

In this post, I will demonstrate how to create a WCF transport extension over this memory message bus, for the most common MEP – Request Reply mode. Before we go to the implementation, let recap the WCF transport mode.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scaling-out-Your-Services-by-Message-Bus_E7A2/image_2.png)

The figure above shows the classes that introduced for the request-reply MEP.

First of all, we need a dedicate binding for this transport. As I said in the [previous post](http://geekswithblogs.net/shaunxu/archive/2012/03/23/scaling-out-your-services-by-message-bus-based-wcf-transport-extension.aspx" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank), the binding takes the responsible for creating and adding all necessary binding elements into the channel stack. There could be many binding elements in the stack but the encoding and transport are mandatory. The encoding binding element defines how the message should be serialized and deserialized. And the transport binding element must be the last on in the stack.

As I mentioned, we can leverage the WCF build-in CustomBinding, but for a fully example I will create our own binding in this series.

The transportbinding element is the root of our transport extension, it takes the responsible to create all following stuff and pass all necessary information into the next level. Based on the service or client side, transport binding element will create the instance of channel factory and the channel listener. The binding and transport binding element will be used on both client and server side.

On the client side, the channel factory(actually CanBuildChanelFactaory(…) of BindingElement will be checked) will be used to check if the MEP is support by this transport. And if applicable, it will create the related channel. For example, if the current MEP is request-reply and is supported, then the channel factory will create a request channel.

How many EMPs the transport can support is based on us. That’s to say, we defines which MEP can be supported by our transport. It’s no need to support all MEPs in a transport. For example, the basic HTTP transport doesn’t support duplex MEP, the MSMQ transport only support datagram MEP.

On the server side, the channel listener take the similar responsible as the channel factory on client side. It will create the related channel as well on the server side.

The channels, in this case the client side request channel and server side reply channel, takes the responsible to receive and send proper messages from the underlying transportation, which is our message bus in this case.

## Binding

Our binding class will be responsible for creating all necessary binding elements. In WCF we can define a binding by the code, or by the configuration. But in this example I will only show how to do it by program. And in order to make things as simple as possible, the binding will create only two elements:

* TextMessageEncodingBindingElement: This will help us to encode the WCF message into the plain text, which is very easy to debug and discover what the actual message is.
* MessageBusTransportBindingElement: This is the transport binding element which we will create later.

As I’ve said, since the MessageBusTransportBindingElement is the transport binding element, it must be added at end of all others.

1: public class MessageBusTransportBinding : Binding

2: {

3: private readonly MessageEncodingBindingElement \_messageElement;

4: private readonly MessageBusTransportBindingElement \_transportElement;

5:

6: public MessageBusTransportBinding()

7: : base()

8: {

9: \_messageElement = new TextMessageEncodingBindingElement();

10: \_transportElement = new MessageBusTransportBindingElement();

11: }

12:

13: public override BindingElementCollection **CreateBindingElements**()

14: {

15: var elements = new BindingElementCollection();

16: elements.Add(\_messageElement);

17: // the transport binding element must be the last one

18: elements.Add(\_transportElement);

19: return elements.Clone();

20: }

21: }

Our MessageBusTransportBinding class inherits from the System.ServiceModel.Channels.Binding class, which is the base class for any bindings, such as basic HTTP, NET.TCP, etc.. And in the constructor we initialize the TextMessageEncodingBindingElement and MessageBusTransportBindingElement. When the WCF creates this binding it will invoke the CreateBindingElements method, and in this method we add these 2 binding elements.

Highlight once again, the transport binding element, MessageBusTransportBindingElement in our example, MUST be added at the end of the list of the binding elements.

Another property that must be implemented for a binding is the Scheme. If you are familiar with web development you would know that the scheme is the first part of the URL, which defines the protocol of your connection. WCF absorbs this concept and for each binding it must have a predefined scheme. Since the scheme is related with the underlying transportation, so it will be defined in the transport binding element. Hence in the binding layer we just need to return the transport binding element scheme back.

1: public override string Scheme

2: {

3: get

4: {

5: return \_transportElement.Scheme;

6: }

7: }

Since we need to make the transport binding element to be able to operate our message bus, we also need to pass the bus instance into the binding and send to the underlying object. So the constructor need to have a parameter to accept the bus object and pass it to the transport binding element. So the constructor of our binding should be like this.

1: public MessageBusTransportBinding(IBus bus)

2: : base()

3: {

4: \_messageElement = new TextMessageEncodingBindingElement();

5: \_transportElement = new MessageBusTransportBindingElement(bus);

6: }

OK, now we have the binding ready, the full code should be like this below and the next step is to create our own transport binding element.

1: public class MessageBusTransportBinding : Binding

2: {

3: private readonly MessageEncodingBindingElement \_messageElement;

4: private readonly MessageBusTransportBindingElement \_transportElement;

5:

6: public MessageBusTransportBinding(IBus bus)

7: : base()

8: {

9: \_messageElement = new TextMessageEncodingBindingElement();

10: \_transportElement = new MessageBusTransportBindingElement(bus);

11: }

12:

13: public override BindingElementCollection CreateBindingElements()

14: {

15: var elements = new BindingElementCollection();

16: elements.Add(\_messageElement);

17: // the transport binding element must be the last one

18: elements.Add(\_transportElement);

19: return elements.Clone();

20: }

21:

22: public override string Scheme

23: {

24: get

25: {

26: return \_transportElement.Scheme;

27: }

28: }

29: }

## Transport Binding Element

The transport binding element inherits from System.ServiceModel.Channels.TransportBindingElement base class and it has three main responsibilities:

* Define the transport scheme.
* Determined what kind of MEP it supports, which means what kind of channels it can create.
* Create the relevant channel factory and channel listener based on the current MEP.

Let’s implement them one by one. Firstly, we must define the scheme of this transport, at it will be the scheme of the endpoint address of all services that use our transport. In our example I would like to use “net.bus” as the scheme, so in the future the services on our transport would be using the endpoint likes “net.bus://localhost/MySameplService”.

1: public class MessageBusTransportBindingElement : TransportBindingElement

2: {

3: public const string CST\_SCHEME = "net.bus";

4:

5: public override string Scheme

6: {

7: get

8: {

9: return CST\_SCHEME;

10: }

11: }

12: }

There are two generic methods we must implement to determined what kind of MEP it supports, which are the CanBuildChannelFactory<TChannel> and CanBuildChannelListener<TChannel>. We check the type parameter TChannel and based on its type value, we return true or false back. Since currently we only support request-reply mode, so it will return true only if the TChannel is IRplyChannel on the server side (CanBuildChannelListener) and IRequestChannel on the client side (CanBuildChannelFactory).

1: public override bool CanBuildChannelFactory<TChannel>(BindingContext context)

2: {

3: return typeof(TChannel) == typeof(IRequestChannel);

4: }

5:

6: public override bool CanBuildChannelListener<TChannel>(BindingContext context)

7: {

8: return typeof(TChannel) == typeof(IReplyChannel);

9: }

Then the methods BuildChannelFactory<TChannel> and BuildChannelListener<TChannel> would be the place where we will create the proper channel instance. The BuildChannelFactory<TChannel> will create the channels on the client side, which based on the TChannel, while the BuildChannelListener<TChannel> will create the channels on the server side.

1: public override IChannelFactory<TChannel> BuildChannelFactory<TChannel>(BindingContext context)

2: {

3: if (context == null)

4: {

5: throw new ArgumentNullException("context");

6: }

7: if (!CanBuildChannelFactory<TChannel>(context))

8: {

9: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel factory.", typeof(TChannel).Name));

10: }

11:

12: if (typeof(TChannel) == typeof(IRequestChannel))

13: {

14: return (IChannelFactory<TChannel>)(object)new MessageBusRequestChannelFactory(this, context);

15: }

16: else

17: {

18: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel listener.", typeof(TChannel).Name));

19: }

20:

21: }

22:

23: public override IChannelListener<TChannel> BuildChannelListener<TChannel>(BindingContext context)

24: {

25: if (context == null)

26: {

27: throw new ArgumentNullException("context");

28: }

29: if (!CanBuildChannelListener<TChannel>(context))

30: {

31: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel listener.", typeof(TChannel).Name));

32: }

33:

34: if (typeof(TChannel) == typeof(IReplyChannel))

35: {

36: return (IChannelListener<TChannel>)(object)new MessageBusReplyChannelListener(this, context);

37: }

38: else

39: {

40: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel listener.", typeof(TChannel).Name));

41: }

42: }

Since we also need the bus object to be passed into the channel factory and channel listener we will add a private member to store it. And also the base class TransportBindingElement needs some other abstract members we must implement. These are fairly easy I don’t want to explain here, the full code of our transport binding element would be like this.

1: public class MessageBusTransportBindingElement : TransportBindingElement

2: {

3: public const string CST\_SCHEME = "net.bus";

4:

5: private readonly IBus \_bus;

6:

7: public IBus Bus

8: {

9: get

10: {

11: return \_bus;

12: }

13: }

14:

15: public override string Scheme

16: {

17: get

18: {

19: return CST\_SCHEME;

20: }

21: }

22:

23: public MessageBusTransportBindingElement(IBus bus)

24: : base()

25: {

26: \_bus = bus;

27: }

28:

29: public MessageBusTransportBindingElement(MessageBusTransportBindingElement other)

30: : base(other)

31: {

32: \_bus = other.\_bus;

33: }

34:

35: public override BindingElement Clone()

36: {

37: return new MessageBusTransportBindingElement(this);

38: }

39:

40: public override bool CanBuildChannelFactory<TChannel>(BindingContext context)

41: {

42: return typeof(TChannel) == typeof(IRequestChannel);

43: }

44:

45: public override bool CanBuildChannelListener<TChannel>(BindingContext context)

46: {

47: return typeof(TChannel) == typeof(IReplyChannel);

48: }

49:

50: public override IChannelFactory<TChannel> BuildChannelFactory<TChannel>(BindingContext context)

51: {

52: if (context == null)

53: {

54: throw new ArgumentNullException("context");

55: }

56: if (!CanBuildChannelFactory<TChannel>(context))

57: {

58: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel factory.", typeof(TChannel).Name));

59: }

60:

61: if (typeof(TChannel) == typeof(IRequestChannel))

62: {

63: return (IChannelFactory<TChannel>)(object)new MessageBusRequestChannelFactory(this, context);

64: }

65: else

66: {

67: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel listener.", typeof(TChannel).Name));

68: }

69:

70: }

71:

72: public override IChannelListener<TChannel> BuildChannelListener<TChannel>(BindingContext context)

73: {

74: if (context == null)

75: {

76: throw new ArgumentNullException("context");

77: }

78: if (!CanBuildChannelListener<TChannel>(context))

79: {

80: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel listener.", typeof(TChannel).Name));

81: }

82:

83: if (typeof(TChannel) == typeof(IReplyChannel))

84: {

85: return (IChannelListener<TChannel>)(object)new MessageBusReplyChannelListener(this, context);

86: }

87: else

88: {

89: throw new ArgumentException(string.Format("Unsupported channel type: {0} for the channel listener.", typeof(TChannel).Name));

90: }

91: }

92: }

The binding and the transport binding element will be created and used on both server and client. And after the transport binding element, when the channel factory and channel listener was created, our code will be split to server and client. Some of the following classes should be run on the server side while other on the client side, based on what MEP and what channel it is.

## Client: RequestChannelFactory and RequestChannel

The request channel factory will be created by the transport binding element, if the current channel is IRequestChannl on the client side. It inherits from the base class ChannelFactoryBase<IRequestChannel> and has only one responsibility: create the request channel.

Besides this main responsibility we also need some extra work to make our life easy. Since the underlying channel will communicate with the actual bus, we need the MessageEncoderFactory and BufferManager to read, write, encode and decode the message. These objects can be created from our encoder binding element.

1: public class MessageBusRequestChannelFactory : ChannelFactoryBase<IRequestChannel>

2: {

3: private readonly BufferManager \_bufferManager;

4: private readonly MessageEncoderFactory \_encoderFactory;

5: private readonly long \_maxReceivedMessageSize;

6:

7:

8: public MessageBusRequestChannelFactory(MessageBusTransportBindingElement transportElement, BindingContext context)

9: : base(context.Binding)

10: {

11: \_bufferManager = BufferManager.CreateBufferManager(transportElement.MaxBufferPoolSize, int.MaxValue);

12: var encodingElement = context.Binding.Elements.Find<MessageEncodingBindingElement;

13: if (encodingElement == null)

14: {

15: \_encoderFactory = (new TextMessageEncodingBindingElement()).CreateMessageEncoderFactory();

16: }

17: else

18: {

19: \_encoderFactory = encodingElement.CreateMessageEncoderFactory();

20: }

21: \_maxReceivedMessageSize = transportElement.MaxReceivedMessageSize;

22: }

23: }

The MessageVersion property defines which SOAP version and addressing version we are going to use in our transport. Just use the default value would be OK. In order to make the inner channel could be able to use the message bus, we also need to pass the bus object from the transport binding element. So the full code of our RequestChannlFactory would be like this.

1: public class MessageBusRequestChannelFactory : ChannelFactoryBase<IRequestChannel>

2: {

3: private readonly BufferManager \_bufferManager;

4: private readonly MessageEncoderFactory \_encoderFactory;

5: private readonly long \_maxReceivedMessageSize;

6: private readonly IBus \_bus;

7:

8: public MessageBusRequestChannelFactory(MessageBusTransportBindingElement transportElement, BindingContext context)

9: : base(context.Binding)

10: {

11: \_bufferManager = BufferManager.CreateBufferManager(transportElement.MaxBufferPoolSize, int.MaxValue);

12: var encodingElement = context.Binding.Elements.Find<MessageEncodingBindingElement;

13: if (encodingElement == null)

14: {

15: \_encoderFactory = (new TextMessageEncodingBindingElement()).CreateMessageEncoderFactory();

16: }

17: else

18: {

19: \_encoderFactory = encodingElement.CreateMessageEncoderFactory();

20: }

21: \_maxReceivedMessageSize = transportElement.MaxReceivedMessageSize;

22: \_bus = transportElement.Bus;

23: }

24:

25: public MessageVersion MessageVersion

26: {

27: get

28: {

29: return MessageVersion.Default;

30: }

31: }

32:

33: public long MaxReceivedMessageSize

34: {

35: get

36: {

37: return \_maxReceivedMessageSize;

38: }

39: }

40:

41: protected override IRequestChannel OnCreateChannel(System.ServiceModel.EndpointAddress address, Uri via)

42: {

43: return new MessageBusRequestChannel(\_bufferManager, \_encoderFactory, this, address, via, \_bus);

44: }

45:

46: protected override IAsyncResult OnBeginOpen(TimeSpan timeout, AsyncCallback callback, object state)

47: {

48: throw new NotImplementedException();

49: }

50:

51: protected override void OnEndOpen(IAsyncResult result)

52: {

53: throw new NotImplementedException();

54: }

55:

56: protected override void OnOpen(TimeSpan timeout)

57: {

58: }

59: }

Now we have entered the last thing on the client side, the request channel. As the lowest level in the transport structure, the channel should send and receive the actual message to the actual transportation, which is our message bus in this case. All channels are inherited from ChannelBase class, and implement the related channel interface. Since we are going to implement our request channel, so it should be inherited from the ChannelBase and the IRequestChannel.

1: public class MessageBusRequestChannel : ChannelBase, IRequestChannel

2: {

3: }

The ChannelBase abstract class need us to implement some abstract methods, which will be invoked when the channel is opened, closed and abort. In order to make our sample clear I just implement a blank OnOpen method, which will be invoked when the channel is opened but do nothing.

1: public class MessageBusRequestChannel : ChannelBase, IRequestChannel

2: {

3: protected override void OnAbort()

4: {

5: throw new NotImplementedException();

6: }

7:

8: protected override IAsyncResult OnBeginClose(TimeSpan timeout, AsyncCallback callback, object state)

9: {

10: throw new NotImplementedException();

11: }

12:

13: protected override IAsyncResult OnBeginOpen(TimeSpan timeout, AsyncCallback callback, object state)

14: {

15: throw new NotImplementedException();

16: }

17:

18: protected override void OnClose(TimeSpan timeout)

19: {

20: throw new NotImplementedException();

21: }

22:

23: protected override void OnEndClose(IAsyncResult result)

24: {

25: throw new NotImplementedException();

26: }

27:

28: protected override void OnEndOpen(IAsyncResult result)

29: {

30: throw new NotImplementedException();

31: }

32:

33: protected override void OnOpen(TimeSpan timeout)

34: {

35: }

36: }

The IRequestChannel defines all operations to send a request message from the client to server. In order to initialize the WCF message, add the message into the bus. So we will add some private members to helps us to operate the message.

1: private readonly IBus \_bus;

2: private readonly Uri \_via;

3: private readonly EndpointAddress \_remoteAddress;

4: private readonly object \_aLock;

5:

6: public MessageBusRequestChannel(

7: BufferManager bufferManager, MessageEncoderFactory encoder, MessageBusRequestChannelFactory parent,

8: EndpointAddress remoteAddress, Uri via, IBus bus)

9: : base(parent)

10: {

11: \_via = via;

12: \_remoteAddress = remoteAddress;

13: \_bus = bus;

14: \_aLock = new object();

15: }

Then what we need to do is to implement the two Request methods, one with a timeout parameter the other isn’t. The method without the timeout parameter could be just invoking the other one with the default send timeout value.

1: public Message Request(Message message)

2: {

3: return Request(message, DefaultSendTimeout);

4: }

For the other method, this is where we need to send the client request message, which comes from the method parameter, into the message bus. First of all we’d better check if the current channel is available and opened. Just invoke a based method.

1: public Message Request(Message message, TimeSpan timeout)

2: {

3: ThrowIfDisposedOrNotOpen();

4: }

And then what we need to do is to

* Unbox the WCF message and convert it into string, which can be sent into out message bus.
* Send the message into bus.
* Wait for the reply message comes from the server side.
* Box the message into WCF message format and return back.

Since we are using in memory bus, in fact we don’t need to convert the WCF message into string. We can send the message object directly into the bus. But in a real distributed scenario we cannot do this as the server and client are in different machine. So I’m converting them to string and show how to use the buffer manager and encoder to conversion between string and WCF message.

We can simply get the content of a WCF message by using its ToString() method, but this is not the best way. In WCF when we need to do the message encoding and decoding, we must use the MessageEncoder, which passed from our binding, and leverage the BufferManager to maximize performance and memory usage. If we go back to the constructor of our channel class you can see we had created the proper encoder from the parameter. So in the Request method we will use this encoder and the buffer manager to read the message and convert it into string. For more information about the encoder and the buffer manager please have a look on the MSDN documents [here](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.messageencoder.aspx" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank) and [here](http://msdn.microsoft.com/en-us/library/system.servicemodel.channels.buffermanager.aspx" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank).

1: // unbox the message into string that will be sent into the bus

2: ArraySegment<byte> buffer;

3: string content;

4: using (message)

5: {

6: \_remoteAddress.ApplyTo(message);

7: buffer = \_encoder.WriteMessage(message, 64 \* 1024, \_bufferManager);

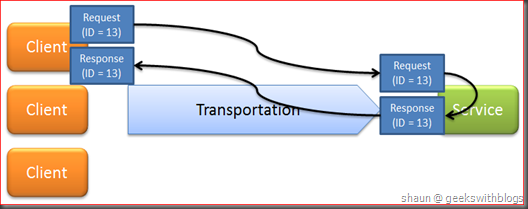
8: }

9: content = Encoding.UTF8.GetString(buffer.Array, buffer.Offset, buffer.Count);

10: \_bufferManager.ReturnBuffer(buffer.Array);

Now we have the message content, then we can send it into the message bus by using the bus object that passed from the constructor. Since the request channel is located on the client side, when invoke the send method of the bus we should tell it this is come from client. And currently we don’t need to tell the server side where the request message come from, so we will leave the last parameter as null.

The SendRequest method will add the message into bus and get a unique ID back. Then the request channel will wait for the reply message, which takes this ID, back into the bus. This unique ID ensure that only this client can receive its own reply message. This is very importance if we have multiple clients are communicating with the same server.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scaling-out-Your-Services-by-Message-Bus_E7A2/image_4.png)

1: // send the message into bus

2: var busMsgId = \_bus.SendRequest(content, true, null);

3: // waiting for the reply message arrive from the bus

4: var replyMsg = \_bus.Receive(false, busMsgId);

Finally, when the request channel received the related reply message, we will use the similar way to create the WCF message based on the content and return back.

1: // box the message from the bus message content and return back

2: var raw = Encoding.UTF8.GetBytes(replyMsg.Content);

3: var data = \_bufferManager.TakeBuffer(raw.Length);

4: Buffer.BlockCopy(raw, 0, data, 0, raw.Length);

5: buffer = new ArraySegment<byte>(data, 0, raw.Length);

6: var reply = \_encoder.ReadMessage(buffer, \_bufferManager);

7: return reply;

Now we had finished all classes and operations on the client side to send the request message and wait for the reply message. The full code of the request channel would be like this.

1: public class MessageBusRequestChannel : ChannelBase, IRequestChannel

2: {

3: private readonly BufferManager \_bufferManager;

4: private readonly MessageEncoder \_encoder;

5:

6: private readonly IBus \_bus;

7: private readonly Uri \_via;

8: private readonly EndpointAddress \_remoteAddress;

9: private readonly object \_aLock;

10:

11: public MessageBusRequestChannel(

12: BufferManager bufferManager, MessageEncoderFactory encoder, MessageBusRequestChannelFactory parent,

13: EndpointAddress remoteAddress, Uri via, IBus bus)

14: : base(parent)

15: {

16: \_bufferManager = bufferManager;

17: \_encoder = encoder.CreateSessionEncoder();

18:

19: \_via = via;

20: \_remoteAddress = remoteAddress;

21: \_bus = bus;

22: \_aLock = new object();

23: }

24:

25: #region IRequestChannel

26:

27: public Uri Via

28: {

29: get

30: {

31: return \_via;

32: }

33: }

34:

35: public IAsyncResult BeginRequest(Message message, TimeSpan timeout, AsyncCallback callback, object state)

36: {

37: throw new NotImplementedException();

38: }

39:

40: public IAsyncResult BeginRequest(Message message, AsyncCallback callback, object state)

41: {

42: throw new NotImplementedException();

43: }

44:

45: public Message EndRequest(IAsyncResult result)

46: {

47: throw new NotImplementedException();

48: }

49:

50: public System.ServiceModel.EndpointAddress RemoteAddress

51: {

52: get { throw new NotImplementedException(); }

53: }

54:

55: public Message Request(Message message, TimeSpan timeout)

56: {

57: ThrowIfDisposedOrNotOpen();

58: lock (\_aLock)

59: {

60: // unbox the message into string that will be sent into the bus

61: ArraySegment<byte> buffer;

62: string content;

63: using (message)

64: {

65: \_remoteAddress.ApplyTo(message);

66: buffer = \_encoder.WriteMessage(message, 64 \* 1024, \_bufferManager);

67: }

68: content = Encoding.UTF8.GetString(buffer.Array, buffer.Offset, buffer.Count);

69: \_bufferManager.ReturnBuffer(buffer.Array);

70: // send the message into bus

71: var busMsgId = \_bus.SendRequest(content, true, null);

72: // waiting for the reply message arrive from the bus

73: var replyMsg = \_bus.Receive(false, busMsgId);

74: // box the message from the bus message content and return back

75: var raw = Encoding.UTF8.GetBytes(replyMsg.Content);

76: var data = \_bufferManager.TakeBuffer(raw.Length);

77: Buffer.BlockCopy(raw, 0, data, 0, raw.Length);

78: buffer = new ArraySegment<byte>(data, 0, raw.Length);

79: var reply = \_encoder.ReadMessage(buffer, \_bufferManager);

80: return reply;

81: }

82: }

83:

84: public Message Request(Message message)

85: {

86: return Request(message, DefaultSendTimeout);

87: }

88:

89: #endregion

90:

91: #region ChannelBase

92:

93: protected override void OnAbort()

94: {

95: throw new NotImplementedException();

96: }

97:

98: protected override IAsyncResult OnBeginClose(TimeSpan timeout, AsyncCallback callback, object state)

99: {

100: throw new NotImplementedException();

101: }

102:

103: protected override IAsyncResult OnBeginOpen(TimeSpan timeout, AsyncCallback callback, object state)

104: {

105: throw new NotImplementedException();

106: }

107:

108: protected override void OnClose(TimeSpan timeout)

109: {

110: throw new NotImplementedException();

111: }

112:

113: protected override void OnEndClose(IAsyncResult result)

114: {

115: throw new NotImplementedException();

116: }

117:

118: protected override void OnEndOpen(IAsyncResult result)

119: {

120: throw new NotImplementedException();

121: }

122:

123: protected override void OnOpen(TimeSpan timeout)

124: {

125: }

126:

127: #endregion

128: }

## Server: ChannelListener, ReplyChannel and RequestContext

In the previous section we finished all stuff on the client side to send a request message to the message bus, and wait for the reply message. On the server side, the binding element will create a channel listener based on the MEP currently used, and create a related reply channel to receive the incoming request message and initialize a request context, which takes the responsible to send the reply message back.

The channel listener is similar with the channel factory, it will initialized from the transport binding element, the BuildChannelListener<TChannel> method.

The reply channel listener inherits from ChannelListenerBase<IReplyChannel>, and in order to make the underlying channel be able to use the buffer manager, encoder and bus to read and write message, we also need to pass them from the binding element, just like what we have done in the channel factory.

1: public class MessageBusReplyChannelListener : ChannelListenerBase<IReplyChannel>

2: {

3: private readonly BufferManager \_bufferManager;

4: private readonly MessageEncoderFactory \_encoderFactory;

5: private readonly Uri \_uri;

6:

7: public override Uri Uri

8: {

9: get

10: {

11: return \_uri;

12: }

13: }

14:

15: public MessageBusReplyChannelListener(MessageBusTransportBindingElement transportElement, BindingContext context)

16: : base(context.Binding)

17: {

18: \_encoderFactory = (new TextMessageEncodingBindingElement()).CreateMessageEncoderFactory();

19: \_bufferManager = BufferManager.CreateBufferManager(transportElement.MaxBufferPoolSize, int.MaxValue);

20: \_uri = new Uri(context.ListenUriBaseAddress, context.ListenUriRelativeAddress);

21: }

22: }

To minimize the implementation, the channel listener only need to implement two methods: OnOpen and OnAcceptChannel. The OnOpen method will be invoked when the listener is opened and began to listen the channel requirement. In this example we don’t need to do anything. The AcceptChannel will be invoked when a channel required, and create a relevant channel object back, based on the MEP it’s. Since we are working on the request-reply mode, we need to reply a IReplyChannel, with the buffer manager, encoder passed though its parameters.

1: protected override IReplyChannel OnAcceptChannel(TimeSpan timeout)

2: {

3: var address = new EndpointAddress(Uri);

4: return new MessageBusReplyChannel(\_bufferManager, \_encoderFactory, address, this);

5: }

6:

7: protected override void OnOpen(TimeSpan timeout)

8: {

9: }

So the full code of the RequestChannelListener would be like this.

1: public class MessageBusReplyChannelListener : ChannelListenerBase<IReplyChannel>

2: {

3: private readonly BufferManager \_bufferManager;

4: private readonly MessageEncoderFactory \_encoderFactory;

5: private readonly Uri \_uri;

6:

7: public override Uri Uri

8: {

9: get

10: {

11: return \_uri;

12: }

13: }

14:

15: public MessageBusReplyChannelListener(MessageBusTransportBindingElement transportElement, BindingContext context)

16: : base(context.Binding)

17: {

18: \_encoderFactory = (new TextMessageEncodingBindingElement()).CreateMessageEncoderFactory();

19: \_bufferManager = BufferManager.CreateBufferManager(transportElement.MaxBufferPoolSize, int.MaxValue);

20: \_uri = new Uri(context.ListenUriBaseAddress, context.ListenUriRelativeAddress);

21: }

22:

23: protected override IReplyChannel OnAcceptChannel(TimeSpan timeout)

24: {

25: var address = new EndpointAddress(Uri);

26: return new MessageBusReplyChannel(\_bufferManager, \_encoderFactory, address, this);

27: }

28:

29: protected override void OnOpen(TimeSpan timeout)

30: {

31: }

32:

33: protected override IAsyncResult OnBeginAcceptChannel(TimeSpan timeout, AsyncCallback callback, object state)

34: {

35: throw new NotImplementedException();

36: }

37:

38: protected override IReplyChannel OnEndAcceptChannel(IAsyncResult result)

39: {

40: throw new NotImplementedException();

41: }

42:

43: protected override IAsyncResult OnBeginWaitForChannel(TimeSpan timeout, AsyncCallback callback, object state)

44: {

45: throw new NotImplementedException();

46: }

47:

48: protected override bool OnEndWaitForChannel(IAsyncResult result)

49: {

50: throw new NotImplementedException();

51: }

52:

53: protected override bool OnWaitForChannel(TimeSpan timeout)

54: {

55: throw new NotImplementedException();

56: }

57:

58: protected override void OnAbort()

59: {

60: throw new NotImplementedException();

61: }

62:

63: protected override IAsyncResult OnBeginClose(TimeSpan timeout, AsyncCallback callback, object state)

64: {

65: throw new NotImplementedException();

66: }

67:

68: protected override IAsyncResult OnBeginOpen(TimeSpan timeout, AsyncCallback callback, object state)

69: {

70: throw new NotImplementedException();

71: }

72:

73: protected override void OnClose(TimeSpan timeout)

74: {

75: throw new NotImplementedException();

76: }

77:

78: protected override void OnEndClose(IAsyncResult result)

79: {

80: throw new NotImplementedException();

81: }

82:

83: protected override void OnEndOpen(IAsyncResult result)

84: {

85: throw new NotImplementedException();

86: }

87: }

The OnAcceptChannel returns a MessageBusReplyChannel which implemented the IReplyChannel and inherits from the ChannelBase. Similar as the request channel, we do not need to do anything in the OnOpen method.

Similar as the channel factory we get the buffer manager, encoder and bus from the upper channel listener.

1: private readonly BufferManager \_bufferManager;

2: private readonly MessageEncoder \_encoder;

3: private readonly EndpointAddress \_localAddress;

4: private readonly object \_aLock;

5:

6: private readonly IBus \_bus;

7:

8: public MessageBusReplyChannel(

9: BufferManager bufferManager, MessageEncoderFactory encoder, EndpointAddress address,

10: MessageBusReplyChannelListener parent,

11: IBus bus)

12: : base(parent)

13: {

14: \_bufferManager = bufferManager;

15: \_encoder = encoder.CreateSessionEncoder();

16:

17: \_localAddress = address;

18: \_bus = bus;

19: \_aLock = new object();

20: }

There are three methods in the IReplyChannel that related with the message receiving: WaitForRequest, ReceiveRequest(TimeSpan) and ReceiveRequest. The WaitForRequest method will return a Boolean when a request message came in, and the two ReceiveRequest read the message and return RequestContext. In our example, since our bus doesn’t support the wait for request feature, it means when receive request we will block the current thread until the request message arrive. So in this case the WaitForRequest will always return true.

And the parameter-less ReceiveRequest method will invoke the other one with the default timeout.

1: public bool WaitForRequest(TimeSpan timeout)

2: {

3: return true;

4: }

5:

6: public RequestContext ReceiveRequest()

7: {

8: return ReceiveRequest(DefaultReceiveTimeout);

9: }

The other ReceiveRequest method is where we need to implement the receive operation. We will firstly check if current channel is available, and receive a message from the bus by calling the Recevie method of the IBus object. If it received a message, then will construct a new RequestContext with the message and the message ID.

1: public RequestContext ReceiveRequest(TimeSpan timeout)

2: {

3: ThrowIfDisposedOrNotOpen();

4: lock (\_aLock)

5: {

6: // receive the request message from the bus

7: var busMsg = \_bus.Receive(true, null);

8: // box the wcf message

9: var raw = Encoding.UTF8.GetBytes(busMsg.Content);

10: var data = \_bufferManager.TakeBuffer(raw.Length);

11: Buffer.BlockCopy(raw, 0, data, 0, raw.Length);

12: var buffer = new ArraySegment<byte>(data, 0, raw.Length);

13: var message = \_encoder.ReadMessage(buffer, \_bufferManager);

14: // initialize the request context and return

15: return new MessageBusRequestContext(message, this, \_bufferManager, \_encoder, \_localAddress, \_bus, busMsg.MessageID);

16: }

17: }

The WCF will take this request context and route it to the proper service, execute your business code. And your business result will be sent back to the request context wrapped into a WCF message into the Reply method. And what we need to do is to send this message into the bus.

The MessageBusRequestContext inherits from RequestContext with many abstract members needs to be implement. And in order to send the reply message into bus we also need the buffer manager, encoder and the bus as well. The implementation of the MessageBusRequestContext, except the Reply method would be like this.

1: public class MessageBusRequestContext : RequestContext

2: {

3: private bool \_aborted;

4: private readonly Message \_message;

5: private readonly MessageBusReplyChannel \_parent;

6: private readonly BufferManager \_bufferManager;

7: private readonly MessageEncoder \_encoder;

8: private readonly EndpointAddress \_address;

9: private readonly object \_aLock;

10: private readonly string \_busMessageId;

11: private readonly IBus \_bus;

12:

13: private CommunicationState \_state;

14:

15: public MessageBusRequestContext(

16: Message message, MessageBusReplyChannel parent,

17: BufferManager bufferManager, MessageEncoder encoder, EndpointAddress address,

18: IBus bus,

19: string relatedTo)

20: {

21: \_aborted = false;

22: \_parent = parent;

23: \_message = message;

24: \_bufferManager = bufferManager;

25: \_encoder = encoder;

26: \_address = address;

27: \_busMessageId = relatedTo;

28: \_bus = bus;

29:

30: \_aLock = new object();

31: \_state = CommunicationState.Opened;

32: }

33:

34: public override void Abort()

35: {

36: lock (\_aLock)

37: {

38: if (\_aborted)

39: {

40: return;

41: }

42: \_aborted = true;

43: \_state = CommunicationState.Faulted;

44: }

45: }

46:

47: public override IAsyncResult BeginReply(Message message, TimeSpan timeout, AsyncCallback callback, object state)

48: {

49: throw new NotImplementedException();

50: }

51:

52: public override IAsyncResult BeginReply(Message message, AsyncCallback callback, object state)

53: {

54: throw new NotImplementedException();

55: }

56:

57: public override void Close(TimeSpan timeout)

58: {

59: lock (\_aLock)

60: {

61: \_state = CommunicationState.Closed;

62: }

63: }

64:

65: public override void Close()

66: {

67: Close(TimeSpan.MaxValue);

68: }

69:

70: public override void EndReply(IAsyncResult result)

71: {

72: throw new NotImplementedException();

73: }

74:

75: public override void Reply(Message message, TimeSpan timeout)

76: {

77: }

78:

79: public override void Reply(Message message)

80: {

81: Reply(message, TimeSpan.MaxValue);

82: }

83:

84: public override Message RequestMessage

85: {

86: get

87: {

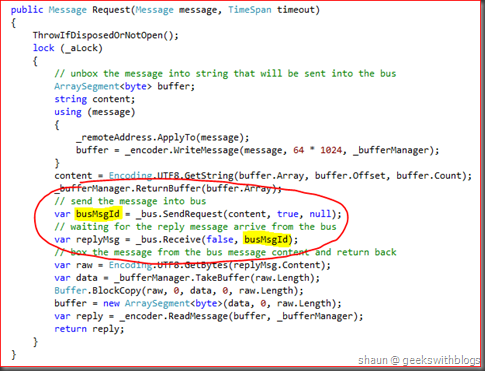
88: return \_message;

89: }

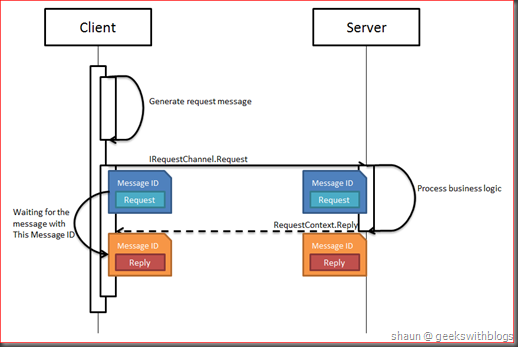
90: }

The reply message from the business logic will be come from the parameter of the Reply method. On the reply channel when we get the message what we need to do is to send the message into our message bus so that the client will receive from its IRequestChannel.Request. So firstly retrieve the content of the reply message, and then send it into message bus.

One thing need to be highlight is that, our reply must be received by the client instance which sent the original request. This is not a major problem when using HTTP or TCP since the underlying connection ensure that the reply will be back to the proper client. But if we are using a message bus and assuming many clients are connecting and sending request messages over the bus, we must have some rules to make sure the reply received by the correct client. If you remembered the code that send the request message we generated a message ID. On the client side after it requested the service, the client will listen the bus only for the message that related with this ID.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scaling-out-Your-Services-by-Message-Bus_E7A2/image_8.png)

And on the server side, we should append this ID onto the reply message as well. The workflow on both server side and client would be like this.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scaling-out-Your-Services-by-Message-Bus_E7A2/image_6.png)

So in the RequestContext.Reply method after got the content from the WCF message we will send it to the bus, with the message ID that passed from the original incoming request message.

1: public override void Reply(Message message, TimeSpan timeout)

2: {

3: // unbox the reply message to string

4: ArraySegment<byte> buffer;

5: string content;

6: using (message)

7: {

8: \_address.ApplyTo(message);

9: buffer = \_encoder.WriteMessage(message, 64 \* 1024, \_bufferManager);

10: }

11: content = Encoding.UTF8.GetString(buffer.Array, buffer.Offset, buffer.Count);

12: \_bufferManager.ReturnBuffer(buffer.Array);

13: // send the reply into bus

14: \_bus.SendReply(content, false, \_busMessageId);

15: }

## Try: Using the Listener, Factory and Channels Directly

We had finished all necessary parts so far and we can start to test our transport. But in this article I don’t want to use the normal way to establish the server and client, which means I will not use ServiceHost, ClientFactory, etc.. We are going to use the inner ChannelListener, ChannelFactory, the IRequestChannel and IReplyChannel to demonstrate what happened in WCF and our transport.

Let’s create a new console application and added the references to the assembly where our transport extension is. Since we are going to use our in memory message bus, this means the server side and client side must be executed in the same process and use the same bus instance. So we will create the bus first.

1: static void Main(string[] args)

2: {

3: var bus = new InProcMessageBus();

4: }

And then we will create the server side stuff. As we know on the server side we need initialize a ChannelListener from the binding, and retrieve a IReplyChannel through the listener’s AcceptChannel method.

1: static void Main(string[] args)

2: {

3: var bus = new InProcMessageBus();

4:

5: // create and open the service listener

6: var svcBinding = new MessageBusTransportBinding(bus);

7: var address = new Uri("net.bus://localhost/sample");

8: var listener = svcBinding.BuildChannelListener<IReplyChannel>(address, new BindingParameterCollection());

9: listener.Open();

10: // create channel and begin to accept

11: var replyChannel = listener.AcceptChannel();

12: replyChannel.Open();

13: }

As you can see we also defined the service listening endpoint, which was “net.bus://localhost/sample”. The scheme of the endpoint must be the same as what we specified in the TransportBindingElement. And use the similar way we created the client side ChannelFactory and the IRequestChannel.

1: static void Main(string[] args)

2: {

3: var bus = new InProcMessageBus();

4:

5: // create and open the service listener

6: ... ...

7: // create channel and begin to accept

8: ... ...

9:

10: // create and open the client factory

11: var cliBinding = new MessageBusTransportBinding(bus);

12: var factory = cliBinding.BuildChannelFactory<IRequestChannel>();

13: factory.Open();

14: // create channel

15: var requestChannel = factory.CreateChannel(new EndpointAddress(address));

16: requestChannel.Open();

17: }

And then here’s a little bit tricky, since the in process message bus force us to execute the server and client side code in the same process we must execute the server side in another thread. So we will create a function that contains the server side code, which includes

* Use the IReplyChannel.WaitForRequest to determined that a request message comes. In our sample since the thread will be blocked by the RecevieRequest method, so the WaitForRequest will always return true.
* Use the IReplyChannel.ReceiveRequest to get the request message in the RequestContext format.
* Load the request content from the message and process the server side business logic. In this example the service will accept a string and reverse it.
* Grab the result into the reply message and invoke the RequestContext.Reply to send it back to the message bus.

1: static void ServerSideProcess(object channel)

2: {

3: var replyChannel = channel as IReplyChannel;

4:

5: while (replyChannel.WaitForRequest(TimeSpan.MaxValue))

6: {

7: using (var context = replyChannel.ReceiveRequest())

8: {

9: using (var message = context.RequestMessage)

10: {

11: Console.WriteLine("Processing request: {0}", message.Headers.Action);

12:

13: // execute the server side business logic

14: var body = message.GetBody<string>();

15: var result = new string(body.Reverse().ToArray());

16:

17: // reply to client

18: var replyMessage = Message.CreateMessage(MessageVersion.Default, "http://sbx.igt.com/SampleService/ReverseResponse", result);

19: context.Reply(replyMessage);

20: }

21: }

22: }

23: }

Back to the Main method we will create a new thread that point it this method we’d just finished and started it with the reply channel object.

1: static void Main(string[] args)

2: {

3: var bus = new InProcMessageBus();

4:

5: // create and open the service listener

6: ... ...

7: // create channel and begin to accept

8: ... ...

9:

10: // create and open the client channel factory

11: ... ...

12: // create channel

13: ... ...

14:

15: // server side: waiting for the request message

16: var serverThread = new Thread(new ParameterizedThreadStart(ServerSideProcess));

17: serverThread.Start(replyChannel);

18: }

Next step is the client side code. We will let the user input any string from the console, wrap it into the request message and sent through the IRequestChannel.Request. It will wait for the reply message came and display the result on the screen. **The full code of this console application would be like this**.

1: class Program

2: {

3: static void Main(string[] args)

4: {

5: var bus = new InProcMessageBus();

6:

7: // create and open the service listener

8: var svcBinding = new MessageBusTransportBinding(bus);

9: var address = new Uri("net.bus://localhost/sample");

10: var listener = svcBinding.BuildChannelListener<IReplyChannel>(address, new BindingParameterCollection());

11: listener.Open();

12: // create channel and begin to accept

13: var replyChannel = listener.AcceptChannel();

14: replyChannel.Open();

15:

16: // create and open the client factory

17: var cliBinding = new MessageBusTransportBinding(bus);

18: var factory = cliBinding.BuildChannelFactory<IRequestChannel>();

19: factory.Open();

20: // create channel

21: var requestChannel = factory.CreateChannel(new EndpointAddress(address));

22: requestChannel.Open();

23:

24: // server side: waiting for the request message

25: var serverThread = new Thread(new ParameterizedThreadStart(ServerSideProcess));

26: serverThread.Start(replyChannel);

27:

28: // client side: invoke the service

29: while (true)

30: {

31: Console.Write("Say something: ");

32: var text = Console.ReadLine();

33: if (string.IsNullOrWhiteSpace(text))

34: {

35: break;

36: }

37:

38: var requestMessage = Message.CreateMessage(MessageVersion.Default, "http://sbx.igt.com/SampleService/Reverse", text);

39: var replyMessage = requestChannel.Request(requestMessage);

40: using (replyMessage)

41: {

42: Console.WriteLine("Processing reply: {0}", replyMessage.Headers.Action);

43: Console.WriteLine("Reply: {0}", replyMessage.GetBody<string>());

44: }

45: }

46: }

47:

48: static void ServerSideProcess(object channel)

49: {

50: var replyChannel = channel as IReplyChannel;

51:

52: while (replyChannel.WaitForRequest(TimeSpan.MaxValue))

53: {

54: using (var context = replyChannel.ReceiveRequest())

55: {

56: using (var message = context.RequestMessage)

57: {

58: Console.WriteLine("Processing request: {0}", message.Headers.Action);

59:

60: // execute the server side business logic

61: var body = message.GetBody<string>();

62: var result = new string(body.Reverse().ToArray());

63:

64: // reply to client

65: var replyMessage = Message.CreateMessage(MessageVersion.Default, "http://sbx.igt.com/SampleService/ReverseResponse", result);

66: context.Reply(replyMessage);

67: }

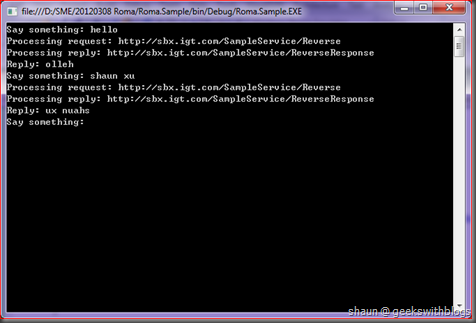
68: }

69: }

70: }

71: }

Let’s start the application and try to input some strings. You can see the message was sent into the message bus and received from the service code, processing the business logic and sent back to the client.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scaling-out-Your-Services-by-Message-Bus_E7A2/image_10.png)

But this is not our final goal. What we wanted to do is to build a transport that the same service can run multiple instances (scaling-out) listening on the same endpoint to accept the clients requests. The sample code below just ran one service instance with one client. Now let’s tweak the application a bit.

I created a new method to establish a new service listener and listen on the message bus in a thread. And in the main method I initialized 3 of them in threads. And in order to show which service thread was processing the request I also tweaked the server side business logic.

1: class Program

2: {

3: static void Main(string[] args)

4: {

5: var bus = new InProcMessageBus();

6: var address = new Uri("net.bus://localhost/sample");

7:

8: // launch multiple services (in threads)

9: LaunchService(bus, address, 1);

10: LaunchService(bus, address, 2);

11: LaunchService(bus, address, 3);

12:

13: // create and open the client factory

14: var cliBinding = new MessageBusTransportBinding(bus);

15: var factory = cliBinding.BuildChannelFactory<IRequestChannel>();

16: factory.Open();

17: // create channel

18: var requestChannel = factory.CreateChannel(new EndpointAddress(address));

19: requestChannel.Open();

20:

21:

22: // client side: invoke the service

23: while (true)

24: {

25: Console.Write("Say something: ");

26: var text = Console.ReadLine();

27: if (string.IsNullOrWhiteSpace(text))

28: {

29: break;

30: }

31:

32: var requestMessage = Message.CreateMessage(MessageVersion.Default, "http://sbx.igt.com/SampleService/Reverse", text);

33: var replyMessage = requestChannel.Request(requestMessage);

34: using (replyMessage)

35: {

36: Console.WriteLine("Processing reply: {0}", replyMessage.Headers.Action);

37: Console.WriteLine("Reply: {0}", replyMessage.GetBody<string>());

38: }

39: }

40: }

41:

42: static void LaunchService(IBus bus, Uri address, int id)

43: {

44: // create and open the service listener

45: var svcBinding = new MessageBusTransportBinding(bus);

46: var listener = svcBinding.BuildChannelListener<IReplyChannel>(address, new BindingParameterCollection());

47: listener.Open();

48: // create channel and begin to accept

49: var replyChannel = listener.AcceptChannel();

50: replyChannel.Open();

51: // server side: waiting for the request message

52: var thread = new Thread((obj) =>

53: {

54: var tuple = obj as Tuple<IReplyChannel, int>;

55: var channel = tuple.Item1;

56: var svcId = tuple.Item2;

57:

58: while (channel.WaitForRequest(TimeSpan.MaxValue))

59: {

60: using (var context = channel.ReceiveRequest())

61: {

62: using (var message = context.RequestMessage)

63: {

64: Console.WriteLine("[ID = {0}]: Processing request: {0}", svcId, message.Headers.Action);

65: // execute the server side business logic

66: var body = message.GetBody<string>();

67: var result = new string(body.Reverse().ToArray());

68: // reply to client

69: var replyMessage = Message.CreateMessage(MessageVersion.Default, "http://sbx.igt.com/SampleService/ReverseResponse", result);

70: context.Reply(replyMessage);

71: }

72: }

73: }

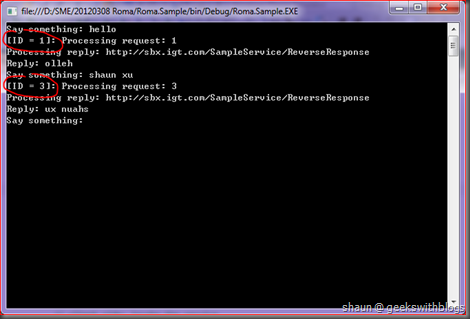
74: });

75: thread.Start(new Tuple<IReplyChannel, int>(replyChannel, id));

76: }

77: }

At this moment if we execute the application and input some strings, we can see that each service thread would be able to process the client request.

[](http://gwb.blob.core.windows.net/shaunxu/Windows-Live-Writer/Scaling-out-Your-Services-by-Message-Bus_E7A2/image_12.png)

I use the service thread to simulate the multiple service instance, due to the in memory message bus limitation. If we are using some standalone message bus, such as TIBCO EMS, Redis or Windows Azure Service Bus Queues & Topics we can run the service in many processes. That would be more convictive.

## Summary

In this article I described how to implement WCF transport extension, in the simplest way. We created the binding, which is the root of the extension. And then the transport binding element, which is the center of the transport. And from the transport binding element, based on the MEP, client or server side we created the channel factory and channel listener. And then from the factory and listener we finally created the channels which takes the responsible to communicate with the transportation, the message bus in our case.

After finished the work below we created a sample application to use the transport and message bus. We didn’t use the normal way to define the services logic, service host and client proxy. We initialize the underlying factory, listener and channels directly.

And finally we simulated the scaling-out on the server side, by running multiple service code in different threads, to prove that our message bus based pulling mode architecture works.

But if we review the sample application you can see that it’s not following the normal WCF pattern. Our service business logic was mixed into our service host and listening code. What we are familiar with when using WCF is to create a service contract interface and service implementation class, and then use the ServiceHost class to host it on a transport.

In the next post I will show you how to modify our transport extension, so the user could be able to define and host the service in the way we are familiar with.

Download the code [here](http://shaunxu.blob.core.windows.net/shared/shaun.xu.roma.p2.zip" \t "http://geekswithblogs.net/shaunxu/archive/2012/03/23/_blank).

Hope this helps,

Shaun

<http://geekswithblogs.net/shaunxu/archive/2012/03/23/scaling-out-your-services-by-message-bus-based-wcf-transport-extension-again.aspx>